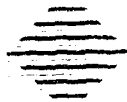


DTIC FILE COPY

Technical Report
CMU/SEI-89-TR-21

②



Carnegie Mellon University
Software Engineering Institute

AD-A219 018

1989 SEI Report on Graduate Software Engineering Education

Mark Ardis, Gary Ford
June 1989

DTIC
ELECTE
MAR 13 1990
S B D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

90 03 12 104

Technical Report

CMU/SEI-89-TR-21

ESD-TR-89-29

June 1989

**1989 SEI Report on Graduate
Software Engineering Education**



Mark Ardis

Video Dissemination Project

Gary Ford

Software Engineering Curriculum Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

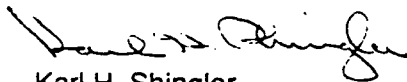
SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1989 by Carnegie Mellon University.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1. Introduction	1
2. The SEI Curriculum Design Workshop	3
2.1. Workshop Organization and Procedures	3
2.2. Discussion	6
3. Curriculum for a Master of Software Engineering Degree	8
3.1. Objectives	8
3.2. Prerequisites	10
3.3. Core Curriculum Content	12
3.4. Curriculum Design	20
3.4.1. Software Systems Engineering	21
3.4.2. Specification of Software Systems	24
3.4.3. Principles and Applications of Software Design	27
3.4.4. Software Generation and Maintenance	30
3.4.5. Software Verification and Validation	35
3.4.6. Software Project Management	38
3.5. Project Experience Component	42
3.6. Electives	44
3.7. Pedagogical Considerations	45
3.8. The Structure of the MSE Curriculum	46
4. Survey of Graduate Degree Programs in Software Engineering	48
5. SEI Graduate Curriculum Test Sites	66
6. Summary and a Look Ahead	67
Appendix 1. An Organizational Structure for Curriculum Content	68
Appendix 2. Bloom's Taxonomy of Educational Objectives	74
Appendix 3. SEI Curriculum Modules and Other Publications	75
Appendix 4. Cumulative Acknowledgements	82
Bibliography	83

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



1989 SEI Report on Graduate Software Engineering Education

Abstract

This annual report on graduate software engineering education describes recent SEI educational activities, including the 1988 SEI Curriculum Design Workshop. A model curriculum for a professional Master of Software Engineering degree is presented, including detailed descriptions of six core courses. Fifteen university graduate programs in software engineering are surveyed.

1. Introduction

An ongoing activity of the SEI Education Program is the development and support of a model graduate curriculum in software engineering. In such a rapidly changing discipline, it is important that such a curriculum be reevaluated and revised frequently in order to ensure that it reflects the state of the art. This report describes our recent efforts toward that end.

To put our recent curriculum efforts in perspective, it is helpful to review the history of SEI curriculum recommendations. In 1985, the staff of the Graduate Curriculum Project developed a strawman description of the important subject areas and possible courses for a professional Master of Software Engineering (MSE) degree. This document was reviewed by the participants at the February 1986 SEI Software Engineering Education Workshop [Gibbs87], who offered numerous suggestions for improvement.

We then wrote a revised version of the document, which was widely circulated for additional comments (see Appendix 4). Those comments were analyzed over the winter of 1986-87, and in May 1987 the SEI published *Software Engineering Education: An Interim Report from the Software Engineering Institute* [Ford87]. This report was our first publication of curriculum recommendations, and it addressed not only curriculum content but also the related curriculum issues of educational objectives, prerequisites, student project work, electives, and resources needed to support the curriculum.

The interim report came to be regarded as a *specification* for an MSE curriculum, because it concentrated on the content of the curriculum rather than how that content might be organized into courses or how those courses might be taught. We expected future work to include curriculum *design* (the organization of that content into meaningful courses), *implementation* (the detailed description of each course by instructors of the course), and *execution*, the process of teaching each course. (We have not yet planned a *validation* effort, though we see the need to do so.)

Two events in 1987 made it clear that a curriculum design was needed immediately. First, the SEI established a new project, the Video Dissemination Project, which would work with

cooperating universities to offer graduate-level software engineering courses on videotape. Second, Carnegie Mellon University made a commitment to establish an MSE program within its newly proposed School of Computer Science. Both of these efforts needed a curriculum, including detailed designs for courses.

In February 1988, the SEI sponsored the Curriculum Design Workshop, whose goal was to design an MSE curriculum that was consistent with the specification in the interim report. The workshop produced designs for six core courses.

During 1988, prototype implementations of two of the core courses were taught by the staff of the Video Dissemination Project. Two additional core courses are being taught in 1989. In addition, two universities, which have been designated as SEI *graduate curriculum test sites*, are teaching courses based on the recommendations of the workshop. The experiences of instructors and students are being collected and will be used to improve the next release of the curriculum recommendations.

Section 2 of this report describes the Curriculum Design Workshop. A summary of our current MSE recommendations, including the six core courses, appears in Section 3. For comparison, the graduate software engineering programs of fifteen universities are surveyed in Section 4. Additional information on the graduate curriculum test sites is presented in Section 5.

Additional background material is presented in the appendices. Appendices 1 and 2 are taken from [Ford87]; they present, respectively, an organizational structure for discussing software engineering curriculum content and a summary of Bloom's taxonomy of educational objectives. Appendix 3 provides short descriptions of SEI publications that support graduate education, and Appendix 4 acknowledges the numerous contributors to the recommendations in this report.

2. The SEI Curriculum Design Workshop

In February 1988 we invited several software engineering educators to an MSE curriculum design workshop. The participants were:

Mark Ardis, *SEI*

Jim Collofello, *Arizona State University*

Lionel Deimel, *SEI*

Dick Fairley, *George Mason University*

Gary Ford, *SEI*

Norm Gibbs, *SEI*

Bob Glass, *SEI*

Harvey Hallman, *SEI*

Tom Kraly, *IBM*

Jeff Lasky, *Rochester Institute of Technology*

Larry Morell, *College of William and Mary*

Tom Piatkowski, *State University of New York at Binghamton*

Scott Stevens, *SEI*

Jim Tomayko, *The Wichita State University*

The stated objective of the workshop was to create descriptions of courses in "sufficient detail." Since the main task was to partition the topics (as defined in the interim report) into courses, enough detail was needed for each course to allow independent implementation of the courses. That is, instructors should be able to prepare and teach their courses in relative isolation, just as software implementors are able to produce their modules independently. Of course, awareness of and cooperation with others is important in both activities. But individuals (instructors or software developers) should feel free to make decisions about every aspect of their product that is not already specified in the design.

2.1. Workshop Organization and Procedures

Since we viewed the previous curriculum description as a specification, we viewed its recommendations as constraints that we must satisfy in our design. Therefore, our first step was to review the specification. Some participants noted that other degree programs were worthy of consideration, but all agreed that the specification was a good starting point for our work.

A major constraint in the interim report was the duration of the program: 30 to 36 semester hours, or about 10 to 12 courses. Of these courses, workshop participants suggested that six or seven would constitute the core material and that three or four would be advanced electives. The remainder of the program would be project work. Because of the limited time available during the workshop (two days), we decided to concentrate exclusively on the design of the core courses.

Other constraints included the assumed prerequisite knowledge of entering students (a BS in computer science, or equivalent knowledge and ability), the expected number of faculty (5 full-time for a program of 20 graduates per year), computing resources, and support staff. We assumed that the resource constraints would be met, and they rarely influenced our design decisions. Instead, we made note of these requirements when elaborating our pedagogical concerns for each course.

Our next step was to examine the 20 *content units* in the specification to try to identify appropriate subject areas. (The content units are summarized in Section 3.3 below.) In addition, we attempted to determine the approximate size of each unit, measured by weeks of class time. A relatively coarse scale was used, having only three points: *small* (1-2 weeks), *medium* (3-6 weeks), and *large* (more than 6 weeks).

We found five natural subject areas of content units, whose working titles were Systems Engineering; Software Design and Specification; Implementation; Verification and Validation; and Control and Management. Although these subject areas resemble the phases of the traditional waterfall life cycle model, we did not intend to advocate any specific model for software development. We do believe, however, that the activities of requirements analysis and specification, design, implementation, verification and validation, and project management are probably elements of any reasonable model. Therefore we believe that these five subject areas are legitimate as well as convenient partitions of the curriculum content.

The five subject areas, the content units in each (numbered as in Section 3.3), and the estimated size of each unit are presented below. Notice that one unit, Software Quality Assurance, appears in two subject areas, making it necessary to divide its material accordingly.

Systems Engineering

- | | |
|---------------------------------|--------|
| 11. Software Operational Issues | small |
| 12. Requirements Analysis | medium |
| 14. System Design | small |
| 18. System Integration | small |
| 20. Human Interfaces | small |

Software Design and Specification

- | | |
|--------------------------------|--------|
| 13. Specification | large |
| 15. Software Design | large |
| 19. Embedded Real-Time Systems | medium |

Implementation

- | | |
|-----------------------------|--------|
| 3. Software Generation | small |
| 4. Software Maintenance | medium |
| 16. Software Implementation | medium |

Verification and Validation

7. Software Quality Issues	medium
8. Software Quality Assurance	medium
17. Software Testing	medium

Control and Management

1. The Software Engineering Process	small
2. Software Evolution	small
5. Technical Communication	medium
6. Software Configuration Management	small
8. Software Quality Assurance	small
9. Software Project Organizational and Management Issues	medium
10. Software Project Economics	small

Four of the five subject areas had an estimated size of 12 to 15 weeks each, which caused us to try to design a single core course for each. The Software Design and Specification subject area appeared to have almost 25 weeks of material, so we thought two courses were warranted.

The workshop then broke into three working groups. The first was charged with designing courses for Systems Engineering and for Software Design and Specification; the second concentrated on Implementation and Verification and Validation, and the third worked on Control and Management. Each group met for two to three hours, and then we all reported our progress in a combined session. This process was iterated twice more in the hope that the boundaries between courses could be clearly drawn, without overlaps or gaps.

The product of the working groups was a set of core courses:

- Software Systems Engineering
- Specification of Software Systems
- Principles and Applications of Software Design
- Software Generation and Maintenance
- Software Verification and Validation
- Software Project Management

For each course we tried to describe the prerequisites, the major and minor topics, the relative duration of topics in the course, the educational objective for each major topic (based on an adaptation of Bloom's taxonomy of educational objectives; see Appendix 2), principal references, and other pedagogical concerns. In some cases, we were able to produce relatively detailed descriptions in the first working group session. For other courses, we barely managed to complete a description after all three sessions. This may reflect the differences in maturity of topics within software engineering. Those topics that have been taught success-

fully for several years were easy to package into courses. Newer topics were more difficult to package.

After the workshop, a subset of the participants prepared more detailed draft descriptions of the courses. Each of the courses was reviewed for internal consistency and for its contribution to the overall integrity of the curriculum. The current versions of these course descriptions appear in Section 3.4. Although the participants were given an opportunity to review intermediate forms of this report, the current authors take responsibility for any errors introduced during its preparation.

2.2 Discussion

At first glance, the required courses appear to follow a traditional waterfall life cycle model: requirements, specification, design, implementation, and testing (with project management added to complete the set of courses). However, the courses are not based on that assumption. Instead, the division of topics into courses emphasizes different skills required of students. For example, requirements analysis depends on communication skills (needed for interviewing users) that are not used in implementation. Software engineers may have to perform requirements analysis concurrently with implementation (e.g., as a prototyping activity), but they can best learn the skills independently.

There are no prerequisite relationships among the required courses. On the other hand, some courses depend critically on courses outside of the curriculum. For example, the Software Specification course and the Verification and Validation course require knowledge of discrete mathematics.

The unit on technical communication does not appear in the six core courses. We recommend that it be integrated into all the courses at appropriate places. For example, oral presentation skills can be taught along with software technical reviews, and writing skills can be taught in the first course where significant documents are required. These skills should be reinforced throughout the curriculum by requiring the students to produce written documents and to make oral presentations. In the past, many instructors in the sciences and engineering have shown a reluctance to make technical communication a factor in student evaluations and grades. Because of its importance in software engineering, we strongly urge instructors to make it an integral part of all appropriate courses.

We spent very little time discussing project work, though we assumed that it would be part of the curriculum. The interim report specification recommended that 30% of the program be devoted to this kind of activity. We noted that some of the required courses include a semester-long project and that the equivalent of two additional semester-long project courses were appropriate. Project work might be done in conjunction with required or elective courses, or as independent coursework.

Very little time was spent discussing elective courses. We assumed that a variety of appropriate courses would be offered and that students would take three of them. In some cases we limited the amount of time allocated to a topic in a required course (in order to allow more time for other, equally important topics), noting that more advanced coverage could be given

in an elective course in that area. We did, however, make some recommendations for the types of electives that should be offered:

- Electives in software engineering subjects, such as software development environments, are clearly appropriate.
- Electives in computer science topics, such as database systems, are probably appropriate, especially if they emphasize application and evaluation.
- Electives in systems engineering are probably appropriate.

3. Curriculum for a Master of Software Engineering Degree

The academic community distinguishes two master's level technical degrees. The Master of Science in *Discipline* is a research-oriented degree, and often leads to doctoral study. The Master of *Discipline* is a terminal professional degree intended for a practitioner who will be able to rapidly assume a position of substantial responsibility in an organization. The former degree often requires a thesis, while the latter requires a project or practicum as a demonstration of the level of knowledge acquired. The Master of Business Administration (MBA) degree is perhaps the most widely recognized example of a terminal professional degree.

The SEI was chartered partly in response to the perceived need for a greatly increased number of highly skilled software engineers. It is our belief that this need can be best addressed by encouraging and helping academic institutions to offer a Master of Software Engineering (MSE) degree.

In this section we present our current recommendations for a model MSE curriculum. The curriculum is described in six parts: program objectives, prerequisites, core curriculum content, curriculum design for six core courses, the project experience component, and electives. These are followed by short discussions of pedagogical concerns and the overall structure of the curriculum. These recommendations continue to evolve, and we expect to publish updated versions annually.

3.1. Objectives

The goal of the MSE degree program is to produce a software engineer who can rapidly assume a position of substantial responsibility within an organization. To achieve this goal, we propose a curriculum designed to give the student a body of knowledge that includes balanced coverage of the software engineering process activities, their aspects, and the products they produce (see Appendix 1 for definitions of the terms *activity*, *aspect*, and *product* as used here), along with sufficient experience to bridge the gap between undergraduate programming and professional software engineering.

Specific educational objectives are summarized below; they appear in greater detail in the descriptions of individual curriculum units in the core curriculum content section (Section 3.3). We describe them using a taxonomy adapted from [Bloom56], which has six levels of objectives: knowledge, comprehension, application, analysis, synthesis, and evaluation. (See Appendix 2 for a brief description of this taxonomy.)

Knowledge: In addition to knowledge about all the material described in the subsequent paragraphs, students should be aware of the existence of models, representations, methods, and tools other than those they learn to use in their own studies. Students should be aware that there is always more to learn, and that they will encounter more in their professional careers, whatever they may have learned in school.

Comprehension: The students should understand the software engineering process, both in the sense of abstract models and in the various instances of the process as practiced in industry. They should understand the activities and aspects of the process. They should understand the issues (sometimes called the *software crisis*) that are motivating the growth and evolution of the software engineering discipline. They should understand the differences between academic or personal programming and software engineering; in particular, they should understand that software engineering involves the production of software systems under the constraints of the control and management activities. They should understand a reasonable set of principles, models, representations, methods, and tools, and the role of analysis and evaluation in software engineering. They should understand the existing design paradigms for well-understood systems, such as compilers. They should know of the existence and comprehend the content of appropriate standards. They should understand the fundamental economic, legal, and ethical issues of software engineering.

Application: The students should be able to apply fundamental principles in the performance of the various activities. They should be able to apply appropriate formal methods to achieve results. They should be able to use appropriate tools covering all activities of the software process. They should be able to collect appropriate data for project management purposes, and for analysis and evaluation of both the process and the product. They should be able to execute a plan, such as a test plan, a quality assurance plan, or a configuration management plan; this includes the performance of various kinds of software tests. They should be able to apply documentation standards in the production of all kinds of documents.

Analysis: The students should be able to participate in technical reviews and inspections of various software work products, including documents, plans, designs, and code. They should be able to analyze the needs of customers.

Synthesis: The students should be able to perform the activities leading to various software work products, including requirements specifications, designs, code, and documentation. They should be able to develop plans, such as project plans, quality assurance plans, test plans, and configuration management plans. They should be able to design data for and structures of software tests. They should be able to prepare oral presentations, and to plan and lead software technical reviews and inspections.

Evaluation: The students should be able to evaluate software work products for conformance to standards. They should know appropriate qualitative and quantitative measures of software products, and be able to use those measures in evaluation of products, as in the evaluation of requirements specifications for consistency and completeness, or the measurement of performance. They should be able to perform verification and validation of software. These activities should consider all system requirements, not just functional and performance requirements. They should be able to apply and validate predictive models, such as those for software reliability or project cost estimation. They should be able to evaluate new technologies and tools to determine which are applicable to their own work.

The word *appropriate* occurs several times in the objectives above. The software engineering discipline is new and changing, and there is not a consensus on the best set of representations, methods, or tools to use. Each implementation of the MSE curriculum must be structured to match the goals and resources of the school and its students. In subsequent reports,

the SEI will offer recommendations on the most promising methods and technologies for many of the software engineering activities.

3.2. Prerequisites

Although an undergraduate degree in computer science is the "obvious" prerequisite for the MSE degree, we cannot adopt such a simplistic approach to defining essential prerequisites. We do not want to exclude those experienced practitioners who do not have such a degree but still wish to pursue the MSE degree. Furthermore, students with bachelor's degrees in computer science from different schools, or from the same school but five years apart, are likely to have substantially different knowledge. Thus the prerequisites for the MSE degree must be defined carefully, and must be enforceable and enforced.

The primary prerequisite, therefore, is substantial knowledge of programming-in-the-small. This includes a working knowledge of at least one modern, high-level language (for example, Pascal, Modula-2, Ada) and at least one assembly language. Also important is a knowledge of fundamental concepts of programming, including control and data structures, modularity, data abstraction and information hiding, and language implementations (runtime environments, procedure linkage, and memory management). Students should also be familiar with the *tools of the trade*, meaning a user's knowledge (not a designer's knowledge) of computer organization and architecture, operating systems, and typical software tools (such as an editor, assembler, compiler, and linking loader). A basic knowledge of formal methods and models (and their application) is also essential, including analysis of algorithms and the fundamentals of computability, automata, and formal languages. Most or all of this material is likely to be found in the first three years of an undergraduate computer science degree program.

Knowledge of one or more other major areas of computer science is highly desirable, but not absolutely necessary. Examples are: functional and declarative languages, numerical methods, database systems, compiler construction, computer graphics, or artificial intelligence. This material is usually found in senior-level electives in a computer science degree program. Some schools may choose to allow advanced computer science courses as electives in the MSE program. Knowledge of major applications areas in the sciences and engineering may also be useful.

The mathematics prerequisites are those commonly required in an undergraduate computer science degree: discrete mathematics and some calculus. Some software engineering topics may require additional mathematical prerequisites, such as probability and statistics. A student planning a career in a particular application area may want additional mathematics, such as linear algebra or differential equations, but these are not essential prerequisites for any of the mainstream software engineering courses.

Enforcing the prerequisites can be difficult. A lesson may be learned from experience with master's degree programs in computer science. In the 1960s and 1970s, these programs often served almost exclusively as retraining programs for students with undergraduate degrees in other fields (notably mathematics and engineering) rather than as advanced degree programs for students who already had an undergraduate computer science degree. In several

schools, undergraduate computer science majors were not eligible for the master's program because they had already taken all or nearly all of the courses as undergraduates.

These programs existed because there was a clearly visible need for more programmers and computer scientists, and the applicants for these programs did not want a second bachelor's degree. There were not enough applicants who already had a computer science degree to permit enforcement of substantial prerequisites.

For the proposed MSE program to achieve its goals, it must take students a great distance beyond the undergraduate computer science degree. This, in turn, requires that students entering the program have approximately that level of knowledge. Because of the widely varying backgrounds of potential students, their level of knowledge is very difficult to assess. Standardized examinations, such as the Graduate Record Examination in Computer Science, provide only part of the solution.

We recommend that schools wishing to establish the MSE program consider instituting a *leveling* or *immigration* course to help establish prerequisite knowledge. Such a course rarely fits into the normal school calendar. Rather, it is an intensive two to four week course that is scheduled just before or just after the start of the school year. (However, Texas Christian University has tried a full-semester leveling course; see [Comer86]). Students receive up to 20 hours a week of lectures summarizing all of the prerequisite material. The value of this course is not that the students become proficient in all the material, but that they become aware of deficiencies in their own preparation. Self-study in parallel with the first semester's courses can often remove most of these deficiencies.

Another important part of the immigration course is the introduction of the computing facilities, especially the available software tools, to students with varying backgrounds. Ten to 20 hours each week can be devoted to demonstrations and practice sessions. Because proficiency with tools can greatly increase the productivity of the students in later courses, the time spent in the immigration course can be of enormous value.

Finally, the immigration course can be used to help motivate the study of software engineering. The faculty, and sometimes the students themselves, can present some of their own or others' experiences that led to improved understanding of some of the significant problems of software engineering.

Another kind of prerequisite has been adopted by some MSE programs (including the College of St. Thomas, Seattle University, and Texas Christian University). All require the student to have at least one year of professional experience as a software developer. This requirement has the benefit of giving the students increased motivation for studying software engineering, since it exposes them to the problems of developing systems that are much larger than those seen in the university, and makes them aware of economic and technical constraints on the software development process. On the negative side, schools cannot control the quality of that experience, and students may acquire bad habits that must be unlearned.

We have not found the arguments for an experience prerequisite sufficiently compelling to recommend it for all MSE programs. Other engineering disciplines have successful master's level programs, and even undergraduate programs, without such a prerequisite. Most graduate professional degrees in other disciplines do not require it.

As a discipline grows and evolves, it is a common phenomenon in education for new material to be taught in courses that are simply added onto an existing curriculum. Over time, the new material is assimilated into the curriculum in a process called *curricular compression*. Obsolete material is taken out of the curriculum, but much of the compression is accomplished by reorganization of material to get the most value in the given amount of time.

In a rapidly growing and changing discipline, new material is added faster than curricular compression can accommodate it. In some engineering disciplines, the problem is acute. There is a growing sentiment that the educational requirement for an entry-level position in engineering should be a master's degree or a five-year undergraduate degree [NRC85]. This is especially true for a computer science/software engineering career.

If this level of education is needed for a meaningful entry-level position, then we question the value of sending students out with a bachelor's degree, hoping they will return sometime later for a software engineering degree. The professional experience achieved during that time will not necessarily be significant. Also, the percentage of students intending to return to school who actually do return declines rapidly as time since graduation increases. Therefore, we believe that an MSE curriculum structured to follow immediately after a good undergraduate curriculum offers the best chance of achieving the goals of rapid increases in the quality and quantity of software engineers. Of course, such a program does not preclude admission of students with professional experience.

We do recognize that work experience can be valuable. The experience component of the MSE curriculum, which is discussed later in this report, might be structured to include actual work experience. It may be that the overall educational experience is significantly enhanced if the work component is a coordinated part of the program rather than an interlude between undergraduate and graduate studies.

We also recognize that we must motivate many of the activities in the software engineering process. We see a great need to raise the level of awareness on the part of both students and educators of the differences between undergraduate programming and professional software engineering. The SEI Education Program is working at the undergraduate level to help accomplish this.

3.3. Core Curriculum Content

Software engineering is a broad and diverse discipline. To facilitate discussions of the content of software engineering curricula, we have found it helpful to develop an organizational structure for the discipline; this is presented in Appendix 1. A brief look at this structure is sufficient to conclude that all of software engineering cannot be covered in any curriculum. Selecting a subset of that content appropriate for a particular program and student population is the primary task of a curriculum designer.

We use a broad view of software engineering when choosing the content of the curriculum, and we include several topics that are not part of a typical engineering curriculum. This

statement of the National Research Council about engineering curricula reflects the views of many engineers and educators [NRC85]:

Another element of the problem is that to make the transition from high school graduate to a competent practicing engineer requires more than just the acquisition of technical skills and knowledge. It also requires a complex set of communication, group-interaction, management, and work-orientation skills.

... For example, education for management of the engineering function (as distinct from MBA-style management) is notably lacking in most curricula. Essential non-technical skills such as written and oral communication, planning, and technical project management (including management of the individual's own work and career) are not sufficiently emphasized.

On the other hand, we have narrowed the curriculum by concentrating almost exclusively on *software* engineering (but including some aspects of *systems* engineering) and omitting applications area knowledge. Two major reasons for this are pragmatic: first, the body of knowledge known as software engineering is sufficiently large to require all the available time in a typical master's degree program (and then some); and second, students cannot study all of the applications areas in which they might eventually work. We believe that a student at the graduate level should have acquired the skills for self-education that will allow acquisition of some knowledge in a needed application area.

More important, however, is our strong belief that the variety of applications areas and the level of sophistication of hardware and software systems in each of those areas mandate a development *team* with a substantial range of knowledge and skills among its members. Some members of the team must understand the capabilities of hardware and software components of a system in order to do the highest level specification, while other members must have the skills to design and develop individual components. Software engineers will have responsibility for software components just as electrical, mechanical, or aeronautical engineers, for example, will have responsibility for the hardware components. Scientists, including computer scientists, will also be needed on development teams, and all the scientists and engineers must be able to work together toward a common goal.

The core content of the MSE curriculum is described in *units*, each covering a major topic area, rather than in courses. There are three reasons for this. First, not every topic area contains enough material for a typical university course. Second, combining units into courses can be accomplished in different ways for different organizations. Third, this structure more easily allows each unit to evolve to reflect the changes in software engineering knowledge and practice, while maintaining the stability of the overall curriculum structure.

Because of strong relationships among topics and subtopics, we were unable to find a consensus on an appropriate order of topics. We do, however, recommend a top-down approach that begins with focus on the software engineering process; this overall view is needed to put the individual activities in context. Software management and control activities are presented next, followed by the development activities and product view topics.

Social and ethical issues are also important to the education and development of a professional software engineer. Examples are privacy, data security, and software safety. We do not recommend a course or unit specifically on these issues, but rather encourage instructors to find opportunities to discuss them in appropriate contexts in all courses and to set an

example for students. (The SEI Education Program is currently investigating software engineering ethics as a curriculum topic, and we expect to offer more specific recommendations in a future report.)

The curriculum topics are described below in units of unspecified size. Nearly all have a software engineering activity as the focus. For each, we provide a short description of the subtopics to be covered, the aspects of the activity that are most important, and the educational objectives of the unit. (See Appendix 1 for definitions of the terms *activity* and *aspect* as they are used here.)

1. The Software Engineering Process

- | | |
|-------------------|--|
| <i>Topics</i> | The software engineering process and software products. All of the software engineering activities. The concepts of software process model and software product life cycle model. |
| <i>Aspects</i> | All aspects, as appropriate for the various activities. |
| <i>Objectives</i> | Knowledge of activities and aspects. Some comprehension of the issues, especially the distinctions among the various classes of activities. The students should begin to understand the substantial differences between the programming they have done in an undergraduate program and software engineering as it is practiced professionally. |

2. Software Evolution

- | | |
|-------------------|--|
| <i>Topics</i> | The concept of a software product life cycle. The various forms of a software product, from initial conception through development and operation to retirement. Controlling activities and disciplines to support evolution. Planned and unplanned events that affect software evolution. The role of changing technology. |
| <i>Aspects</i> | Models of software evolution, including development life cycle models such as the waterfall, iterative enhancement, phased development, spiral. |
| <i>Objectives</i> | Knowledge and comprehension of the models. Knowledge and comprehension of the controlling activities. |

3. Software Generation

- | | |
|----------------|---|
| <i>Topics</i> | Various methods of software generation, including designing and coding from scratch, use of program or application generators and very high level languages, use of reusable components (such as mathematical procedure libraries, packages designed specifically for reuse, Ada generic program units, and program concatenation, as with pipes). Role of prototyping. Factors affecting choice of a software generation method. Effects of generation method on other software development activities, such as testing and maintenance. |
| <i>Aspects</i> | Models of software generation. Representations for software generation, including design and implementation languages, very high level languages, and |

application generators. Tools to support generation methods, including application generators.

Objectives Knowledge and comprehension of the various methods of software generation. Ability to apply each method when supported by appropriate tools. Ability to evaluate methods and choose the appropriate ones for each project.

4. Software Maintenance

Topics Maintenance as a part of software evolution. Reasons for maintenance. Kinds of maintenance (perfective, adaptive, corrective). Comparison of development activities during initial product development and during maintenance. Controlling activities and disciplines that affect maintenance. Designing for maintainability. Techniques for maintenance.

Aspects Models of maintenance. Current methods.

Objectives Knowledge and comprehension of the issues of software maintenance and current maintenance practice.

5. Technical Communication

Topics Fundamentals of technical communication. Oral and written communication. Preparing oral presentations and supporting materials. Software project documentation of all kinds.

Aspects Principles of communication. Document preparation tools. Standards for presentations and documents.

Objectives Knowledge of fundamentals of technical communication and of software documentation. Application of fundamentals to oral and written communications. Ability to analyze, synthesize, and evaluate technical communications.

6. Software Configuration Management

Topics Concepts of configuration management. Its role in controlling software evolution. Maintaining product integrity. Change control and version control. Organizational structures for configuration management.

Aspects Fundamental principles. Tools (such as *sccs* or *rcs*). Documentation, including configuration management plans.

Objectives Knowledge and comprehension of the issues. Ability to apply the knowledge to develop a configuration management plan and to use appropriate tools.

7. Software Quality Issues

Topics Definitions of quality. Factors affecting software quality. Planning for quality. Quality concerns in each phase of a software life cycle, with special emphasis on the specification of the pervasive system attributes. Quality measurement and

standards. Software correctness assessment principles and methods. The role of formal verification and the role of testing.

- Aspects* Assessment of software quality, including identifying appropriate measurements and metrics. Tools to help perform measurement. Correctness assessment methods, including testing and formal verification. Formal models of program verification.
- Objectives* Knowledge and comprehension of software quality issues and correctness methods. Ability to apply proof of correctness methods.

8. Software Quality Assurance

- Topics* Software quality assurance as a controlling discipline. Organizational structures for quality assurance. Independent verification and validation teams. Test and evaluation teams. Software technical reviews. Software quality assurance plans.
- Aspects* Current industrial practice for quality assurance. Documents including quality assurance plans, inspection reports, audits, and validation test reports.
- Objectives* Knowledge and comprehension of quality assurance planning. Ability to analyze and synthesize quality assurance plans. Ability to perform technical reviews. Knowledge and comprehension of the fundamentals of program verification and its role in quality assurance. Ability to apply concepts of quality assurance as part of a quality assurance team.

9. Software Project Organizational and Management Issues

- Topics* Project planning: choice of process model, project scheduling and milestones. Staffing: development team organizations, quality assurance teams. Resource allocation.
- Aspects* Fundamental concepts and principles. Scheduling representations and tools. Project documents.
- Objectives* Knowledge and comprehension of concepts and issues. It is not expected that a student, after studying this material, will immediately be ready to manage a software project.

10. Software Project Economics

- Topics* Factors that affect cost. Cost estimation, cost/benefit analysis, risk analysis for software projects.
- Aspects* Models of cost estimation. Current techniques and tools for cost estimation.
- Objectives* Knowledge and comprehension of models and techniques. Ability to apply the knowledge to tool use.

11. Software Operational Issues

- Topics* Organizational issues related to the use of a software system in an organization. Training, system installation, system transition, operation, retirement. User documentation.
- Aspects* User documentation and training materials.
- Objectives* Knowledge and comprehension of the major issues.

12. Requirements Analysis

- Topics* The process of interacting with the customer to determine system requirements. Defining software requirements. Identifying functional, performance, and other requirements: the pervasive system requirements. Techniques to identify requirements, including prototyping, modeling, and simulation.
- Aspects* Principles and models of requirements. Techniques of requirement identification. Tools to support these techniques, if available. Assessing requirements. Communication with the customer.
- Objectives* Knowledge and comprehension of the concepts of requirements analysis and the different classes of requirements. Knowledge of requirements analysis techniques. Ability to apply techniques and analyze and synthesize requirements for simple systems.

13. Specification

- Topics* Objectives of the specification process. Form, content, and users of specifications documents. Specifying functional, performance, reliability, and other requirements of systems. Formal models and representations of specifications. Specification standards.
- Aspects* Formal models and representations. Specification techniques and tools that support them, if available. Assessment of a specification for attributes such as consistency and completeness. Specification documents.
- Objectives* Knowledge and comprehension of the fundamental concepts of specification. Knowledge of specification models, representations, and techniques, and the ability to apply or use one or more. Ability to analyze and synthesize a specification document for a simple system.

14. System Design

- Topics* The role of system design and software design. How design fits into a life cycle. Software as a component of a system. Hardware versus software tradeoffs for system performance and flexibility. Subsystem definition and design. Design of high-level interfaces, both hardware to software and software to software.

- Aspects* System modeling techniques and representations. Methods for system design, including object-oriented design, and tools to support those methods. Iterative design techniques. Performance prediction.
- Objectives* Comprehension of the issues in system design, with emphasis on engineering tradeoffs. Ability to use appropriate system design models, methods, and tools, including those for specifying interfaces. Ability to analyze and synthesize small systems.

15. Software Design

- Topics* Principles of design, including abstraction and information hiding, modularity, reuse, prototyping. Levels of design. Design representations. Design practices and techniques. Examples of design paradigms for well-understood systems.
- Aspects* Principles of software design. One or more design notations or languages. One or more widely used design methods and supporting tools, if available. Assessment of the quality of a design. Design documentation.
- Objectives* Knowledge and comprehension of one or more design representations, design methods, and supporting tools, if available. Ability to analyze and synthesize designs for software systems. Ability to apply methods and tools as part of a design team.

16. Software Implementation

- Topics* Relationship of design and implementation. Features of modern procedural languages related to design principles. Implementation issues, including reusable components and application generators. Programming support environment concepts.
- Aspects* One or more modern implementation languages and supporting tools. Assessment of implementations: coding standards and metrics.
- Objectives* Ability to analyze, synthesize, and evaluate the implementation of small systems.

17. Software Testing

- Topics* The role of testing and its relationship to quality assurance. The nature of and limitations of testing. Levels of testing: unit, integration, acceptance, etc. Detailed study of testing at the unit level. Formal models of testing. Test planning. Black box and white box testing. Building testing environments. Test case generation. Test result analysis.
- Aspects* Testing principles and models. Tools to support specific kinds of tests. Assessment of testing; testing standards. Test documentation.

Objectives Knowledge and comprehension of the role and limitations of testing. Ability to apply test tools and techniques. Ability to analyze test plans and test results. Ability to synthesize a test plan.

18. System Integration

Topics Testing at the software system level. Integration of software and hardware components of a system. Uses of simulation for missing hardware components. Strategies for gradual integration and testing.

Aspects Methods and supporting tools for system testing and system integration. Assessment of test results and diagnosing system faults. Documentation: integration plans, test results.

Objectives Comprehension of the issues and techniques of system integration. Ability to apply the techniques to do system integration and testing. Ability to develop system test and integration plans. Ability to interpret test results and diagnose system faults.

19. Embedded Real-Time Systems

Topics Characteristics of embedded real-time systems. Existence of hard timing requirements. Concurrency in systems; representing concurrency in requirements specifications, designs, and code. Issues related to complex interfaces between devices and between software and devices. Criticality of embedded systems and issues of robustness, reliability, and fault tolerance. Input and output considerations, including unusual data representations required by devices. Issues related to the cognizance of time. Issues related to the inability to test systems adequately.

Objectives Comprehension of the significant problems in the analysis, design, and construction of embedded real-time systems. Ability to produce small systems that involve interrupt handling, low-level input and output, concurrency, and hard timing requirements, preferably in a high-level language.

20. Human Interfaces

Topics Software engineering factors: applying design techniques to human interface problems, including concepts of device independence and virtual terminals. Human factors: definition and effects of screen clutter, assumptions about the class of users of a system, robustness and handling of operator input errors, uses of color in displays.

Objectives Comprehension of the major issues. Ability to apply design techniques to produce good human interfaces. Ability to design and conduct experiments with interfaces, to analyze the results and use them to improve the design.

3.4. Curriculum Design

The six core courses in the MSE curriculum are:

- Software Systems Engineering
- Specification of Software Systems
- Principles and Applications of Software Design
- Software Generation and Maintenance
- Software Verification and Validation
- Software Project Management

Detailed course descriptions are presented in Sections 3.4.1 to 3.4.6; each is organized into eight parts:

Catalog Description	A short description of the course, similar to that in a college catalog.
Course Objectives	A general statement of educational objectives.
Prerequisites	Knowledge required of students prior to taking this course.
Syllabus	An outline of the topics to be covered in the course, with annotations (in italics) and references. For each major topic, the number of weeks to be devoted to the topic and the educational objective (from Bloom's taxonomy) are noted.
Relevant SEI Curriculum Modules	A list of SEI curriculum modules whose content includes topics from the course.
Pedagogical Concerns	A short discussion of how the course should be taught, suggestions for student projects or exercises, and other information of interest to the instructor.
Comments	Information on the development or philosophy of the course, usually derived from discussions at the curriculum design workshop.
Bibliography	References from the syllabus; usually these are background reading for instructors.

A significant fact about these courses is that there is no prerequisite structure among them. This is primarily a result of the overall program prerequisites. A modern undergraduate curriculum in computer science includes significant coverage of programming-in-the-small, including some simple models of software development. Therefore the MSE core courses constitute a second, substantially more detailed, pass through much of this material. Elective courses can provide a third, still more detailed study of some topics.

The primary consideration in scheduling the courses is that they and the student project work (see Section 3.5) are mutually supportive. For many schools, it is likely that the courses will be offered in "waterfall-model order" since the project proceeds in that order.

3.4.1. Software Systems Engineering

Catalog Description

This course exposes students to the development of software systems at the very highest level. It introduces the system aspect of development and the related tradeoffs required when software and hardware are developed together, especially with respect to user interfaces. It exposes students to requirements analysis and techniques for developing a system from those requirements. System integration and transition into use are also covered.

Course Objectives

After completing this course, students should comprehend the alternative techniques used to specify and design systems of software and hardware components. They should be able to find the data and create a requirements document and to develop a system specification. They should understand the concepts of simulation, prototyping, and modeling. They should know what is needed to prepare a system for delivery to the user and what makes a system usable.

Prerequisites

Students should have knowledge of software life cycle models, computer architectures, and basic statistics.

Syllabus

Wks	Topics and Subtopics (<i>Objective</i>)
-----	---

1	Introduction (<i>Knowledge</i>)
---	-----------------------------------

Students should see the "big picture" in this part of the course. The emphasis should be on how software is only one component of a larger system.

Overview of topics

1	System Specification (<i>Comprehension</i>)
---	---

Contents

Standards

Global issues such as safety, reliability

2	System Design (<i>Comprehension</i>)
---	--

Simulation

Queuing theory

Tradeoffs

Methods (levels, object-oriented, function-oriented)

3	Interfaces (<i>Comprehension</i>)
---	-------------------------------------

Both human interfaces and interfaces to hardware devices should be included. These areas require different skills but are logically combined here to emphasize the notion of encapsulation of software within larger systems.

Human factors

Guidelines

Experiments

Devices

1 System Integration (*Comprehension*)

Students should learn how to perform integration of entire systems, not just software.

Simulation of missing components

System build

5 Requirements Analysis (*Synthesis*)

This is the largest part of the course. Students should learn the interpersonal skills as well as the technical skills necessary to elicit requirements from users. Expression and analysis of requirements are often performed with CASE tools.

Objectives

Interview skills

Needs and task analysis

Prototypes

SADT, RSL (and other specific methods)

1 Operations Requirements (*Comprehension*)

Students should understand and know how to satisfy the other operations requirements of systems, such as training and documentation.

Training

Online help

User documentation

Relevant SEI Curriculum Modules

CM-6 *Software Safety*, Nancy G. Leveson

CM-11 *Software Specification: A Framework*, H. Dieter Rombach

CM-17 *User Interface Development*, Gary Perlman

CM-19 *Software Requirements*, John W. Brackett

Pedagogical Concerns

Case studies should be available as assigned readings. A requirements analysis project should be assigned to students, with topics in the lectures sequenced to match the project

schedule. A user interface prototype project should be assigned, including an exercise in user documentation. The students should give a presentation on their requirements study. An instructor of this course should have experience in requirements analysis and system design.

Comments

We had a great deal of difficulty naming this course. Much of the work that students will perform as exercises and projects deals with requirements analysis. On the other hand, this course attempts to place software in perspective with other elements of systems. The theme of the course is not just requirements analysis, but total systems engineering. We noted that universities often have courses titled "systems engineering" that cover the same topics from an electrical engineering perspective.

An important goal of this course is that students achieve an understanding of the role of software engineering within the larger context of systems engineering. They should understand, for example, that while ensuring that a software system satisfies its specification is a *software* problem, getting the right specification is a *systems* problem. If software does not give the right system behavior, it must be determined whether the software fails to meet the specification or whether the specification does not define the right system behavior. These distinctions are critical as students leave the academic world, where the entire system is often a personal computer, and enter the "real world" of embedded systems.

Bibliography

The bibliography for this course is still being developed. The bibliographies of the SEI curriculum modules listed above will provide good references for much of the course.

3.4.2. Specification of Software Systems

Catalog Description

Specification occurs at many levels in software engineering. High-level specifications often attempt to capture user requirements, while detailed functional specifications often describe implementation decisions. This course covers several different models of and languages for specification of software systems. The role of documents and standards and the notion of traceability between documents are also covered.

Course Objectives

After completing this course, students should be able to write specifications in at least one formal language, analyze specifications for consistency and completeness, trace requirements to parts of functional specifications, and be able to recognize and apply a number of standard paradigms.

Prerequisites

Students should have a working knowledge of set theory, functions and relations, and predicate calculus. They should also have basic knowledge of state machines. A course in discrete mathematics usually satisfies this requirement.

The discussion of the role of specifications presumes some knowledge of the software life cycle. For example, traceability presumes knowledge of requirements, at least at the concept level.

Syllabus

Wks	Topics and Subtopics (<i>Objective</i>)
1	Types of Specification (<i>Comprehension</i>) Functional Non-functional: performance, reliability, quality, usability, etc. <i>Non-functional specifications are notoriously hard to describe precisely. It is important that students know about this topic, though the course will emphasize functional specifications.</i>
5.5	Models and Languages of Specification (<i>Synthesis</i>) <i>It is not possible to teach (or even categorize) all of the competing models and languages. Students should be exposed to several different ways of thinking by studying perhaps four of the models listed below. Only one model and language can be mastered well enough to use in a semester-long project.</i> Axiomatic [Guttag79, Guttag80, Guttag85] State-machine [Parnas72, Bartussek78]

Abstract model [Bjørner78, Bjørner82, Jones86]

Operational [Zave81, Zave82]

Concurrency [Hoare78, Harel87, Peterson77]

5.5 Paradigms (*Application*)

For each specification model there are application domains or solutions well suited to that model. Disciplined use of specification languages, including use of domain-specialized templates, is important. The list of paradigms below is meant to be representative, not exhaustive.

Transformational: refinement of specifications into implementations [Agresti86]

Real-time systems: problems involving the notion of time, concurrency, reliability and performance

Data processing: problems that have "batch" solutions

Expert systems: constraint-based problems

2 Role of Documentation (*Comprehension*)

The types of issues that should be addressed in this topic include: Where do specifications fit into the software life cycle? Who are the participants in the writing and reading of specifications? What restrictions are placed on the format of specifications?

Document classes (e.g., the distinction between C-specs and D-specs)

Standards (e.g., Mil Std 2167A)

Traceability to requirements

Relevant SEI Curriculum Modules

CM-8 *Formal Specification of Software*, Alfs Berztiss

CM-11 *Software Specification: A Framework*, H. Dieter Rombach

CM-16 *Software Development Using VDM*, Jan Storbak Pedersen

The overview module by Rombach (CM-11) provides a good framework for concepts and terminology. The modules by Berztiss (CM-8) and Pedersen (CM-16) each cover one formal method in depth.

Pedagogical Concerns

Students should participate in a semester-long project in order to master at least one method and language. Smaller assignments should be given to reinforce understanding of other languages and models. Case studies are an effective means to show practical examples. Since the students will spend a lot of time with at least one language, tool support is important.

In teaching the paradigms topic, good examples are needed. It would be best to interleave the appropriate paradigms with the models and languages most often used. For example, the state-machine and concurrency models could be illustrated with real-time examples.

Comments

Formal specification languages and methods require appropriate motivation within a software engineering curriculum. We believe that the appropriate paradigms should be used to illustrate the formalisms so that students will appreciate the relative merits of each. Most of the formalisms also require significant investment in technology before they can be used effectively. It is unlikely that students can master several languages and tools within one semester. On the other hand, they need to master at least one technology in order to see its benefits.

Bibliography

- Agresti86 Agresti, W. W. "What Are the New Paradigms?" *Tutorial: New Paradigms for Software Development*, William W. Agresti, ed. IEEE Computer Society, 1986.
- Bartussek78 Bartussek, W., and Parnas, D. L. "Using Assertions About Traces to Write Abstract Specifications for Software Modules." *Proc. of Second Conf. of European Cooperation in Informatics*, 1978.
- Bjørner78 Bjørner, D. "Programming in the Meta-Language: A Tutorial." *The Vienna Development Method: The Meta-Language*, Dines Bjørner, Cliff B. Jones, eds. New York: Springer-Verlag, 1978, 24-217.
- Bjørner82 Bjørner, D., and Jones, C. B. *Formal Specification and Software Development*. Englewood Cliffs, N.J.: Prentice-Hall, 1982.
- Guttag79 Guttag, J. V. "Notes on Type Abstraction." *Proc. SRS Conf.*, 1979.
- Guttag80 Guttag, J. V., and Horning, J. J. "Formal Specification as a Design Tool." *Seventh Symp. Principles of Prog. Lang.* ACM, 1980.
- Guttag85 Guttag, J. V., Horning, J. J., and Wing, J. M. "The Larch Family of Specification Languages." *IEEE Software* 2, 5 (Sept. 1985), 24-36.
- Harel87 Harel, D. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming* 8 (1987), 231-274.
- Hoare78 Hoare, C. A. R. "Communicating Sequential Processes." *Comm. ACM* 21, 8 (Aug. 1978), 666-677.
- Jones86 Jones, C. B. "Systematic Program Development." *Proc. Symposium on Mathematics and Computer Science*, 1986.
- Parnas72 Parnas, D. L. "A Technique for Software Module Specification with Examples." *Comm. ACM* 15, 5 (May 1972), 330-336.
- Peterson77 Peterson, J. L. "Petri Nets." *Computing Surveys* 9, 3 (Sept. 1977), 223-252.
- Zave81 Zave, P., and Yeh, R. T. "Executable Requirements for Embedded Systems." *Proc. Fifth Intl. Conf. Soft. Eng.* New York: IEEE, 1981, 295-304.
- Zave82 Zave, P. "An Operational Approach to Requirements Specification for Embedded Systems." *Trans. Soft. Eng. SE-8*, 3 (May 1982), 250-269.

3.4.3. Principles and Applications of Software Design

Catalog Description

Design is a central activity of software development. This course covers several different methods and languages for expressing designs. The process of design assessment is also covered.

Course Objectives

After completing this course, students should be able to use at least one method to design large systems. They should know how to choose the appropriate method and notation for a problem class, be able to evaluate designs created by others, and comprehend several design paradigms.

Prerequisites

Students should have a good working knowledge of programming-in-the-small. Experience in designing small systems is helpful.

Syllabus

Wks	Topics and Subtopics (<i>Objective</i>)
-----	---

1	Design Principles and Attributes (<i>Comprehension</i>)
---	---

Students should learn the value of a good design and learn how to recognize one when they see it.

Abstraction

Information hiding

Modularity

Cohesion and coupling

5	Design Methods (<i>Evaluation</i>)
---	--------------------------------------

The pedagogical objective for this topic is to reach the evaluation level for one method and the comprehension level for the other methods. It is important that students be exposed to several different models, perhaps four from the following list. At the minimum, students should be exposed to both top-down (decomposition) and bottom-up (composition) methods. Examples of top-down methods are iterative enhancement, SCR, Jackson, and Mills. Examples of bottom-up methods are object-oriented and data abstraction. Since dataflow methods will probably be covered in the Software Systems Engineering course, they do not have to be covered here.

Object-oriented

Data abstraction [Liskov86]

Iterative enhancement [Wirth71, Dijkstra68]

Dataflow [Yourdon79, Gane79]
Program design languages (PDLs)
Software Cost Reduction (SCR) [Parnas85]
Jackson (JSP and JSD) [Jackson75, Jackson83]
Mills [Mills86]

1 Design Verification (*Application*)

Designs should be checked for internal consistency and completeness, and for accuracy in elaborating a functional specification. This is typically done by review.

7 Paradigms (*Comprehension*)

Some design methods work better with particular application domains or problem types. For each method, the appropriate examples should be chosen to illustrate the success of that method. Some examples of these paradigms are:

User interfaces

Examples are problems that require the specification and use of windows, icons, devices, or user interface management systems (UIMS).

Real-time

Examples are problems that include timing constraints, concurrency, interrupts, etc.

Distributed systems

Examples are problems that involve reliability, synchronization, and availability of resources.

Embedded systems

Examples are problems that involve interfaces to hardware devices.

Relevant SEI Curriculum Modules

CM-2 *Introduction to Software Design*, David Budgen
CM-3 *The Software Technical Review Process*, James S. Collofello
CM-16 *Software Development Using VDM*, Jan Storbanks Pedersen

Modules on concurrent programming and design of real-time systems are presently under development.

Pedagogical Concerns

There is a need to compare specific methods (e.g., Jackson, Yourdon, Mills), without advocating the use of one method for all purposes. Students should work on a semester-long team project using one method, but different teams might use different methods. The results of the projects should be assessed by students. Paradigms should be interspersed with lectures on specific methods. Case studies are an effective means to illustrate paradigms.

Comments

Although many design notations are currently taught in software engineering courses, the creative process of design is often neglected. Participants in the curriculum design workshop were unable to recommend an approach to teach this process, but they noted that the instructor's experience and abilities play an important role.

Bibliography

- Dijkstra68 Dijkstra, E. "The Structure of the THE Multiprogramming System." *Comm. ACM* 11, 5 (May 1968), 341-346.
- Gane79 Gane, C., and Sarson, T. *Structured Systems Analysis: Tools and Techniques*. Englewood Cliffs, N.J.: Prentice-Hall, 1979.
- Jackson75 Jackson, M. *Principles of Program Design*. London: Academic Press, 1975.
- Jackson83 Jackson, M. *System Development*. Englewood Cliffs, N.J.: Prentice-Hall, 1983.
- Liskov86 Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development*. New York: McGraw-Hill, 1986.
- Mills86 Mills, H. D., Linger, R. C., and Hevner, A. R. *Principles of Information Systems Analysis and Design*. Academic Press, 1986.
- Parnas85 Parnas, D. L., and Weiss, D. M. "Active Design Reviews: Principles and Practices." *Proc. 8th Intl. Conf. Soft. Eng.* IEEE Computer Society Press, 1985, 132-136.
- Wirth71 Wirth, N. "Program Development by Stepwise Refinement." *Comm. ACM* 14, 4 (Apr. 1971).
- Yourdon79 Yourdon, E., and Constantine, L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

3.4.4. Software Generation and Maintenance

Catalog Description

Software generation is the creation or reuse of software. Software maintenance is the revision of existing software. This course describes techniques for performing each of those activities. Topics include alternatives to coding, language concepts, the role of standards and style, the role of tools, performance analysis, regression analysis, and other maintenance-specific subjects.

Course Objectives

After completing this course, students should know several alternatives for generating code, be able to identify good coding style and practices, and know what features of languages assist or inhibit good coding practices. They should be able to improve the performance of implemented software, be familiar with tools to help coding and maintenance, and understand the tradeoffs in maintaining software from specifications or from code.

Prerequisites

Students taking this course should have created and tested simple programs. Having participated in the development or maintenance of a complex program would be valuable.

Syllabus

Wks	Topics and Subtopics (Objective)
-----	----------------------------------

6	Implementation (Application)
---	------------------------------

Alternatives to conventional coding

This subtopic is intended to broaden the perspectives of students with respect to implementation strategies. There are several ways to reuse existing code, such as incorporating software packages or parts. Code can be generated through the use of fourth generation languages or compilable specifications. Finally, templates or macros can be used to reduce the cost of reproducing similar fragments of code.

Language concepts/constraints

Students need to understand the consequences of choosing a particular programming language. For example, some languages support software engineering principles (e.g., abstract data types), while others do not. If a language does not support a desired practice, then style or discipline must be used to achieve that practice. Some languages more easily support particular design paradigms (e.g., Prolog supports constraint-based designs better than Pascal).

Performance analysis [Bentley82, Bentley86]

Students should be exposed to a wide spectrum of techniques for measuring and improving the performance of programs.

Standards and style

Because there is not universal agreement on coding standards and style, instructors must choose the style to teach. There are several books on coding style. Coding standards are more difficult to obtain, but they can be very valuable teaching aids..

8 Maintenance (Comprehension)

Maintenance activities [Glass81]

This subtopic provides an overview of maintenance activities. Students should appreciate the differences between maintaining and generating software.

- Diagnosing and correcting problems
- Introducing new functionality
- Porting to a new environment
- Reducing maintenance costs, modernizing software

Maintaining software engineered artifacts [Martin83, Clapp81, Parnas79]

There is a difference between maintaining a system for which the history of development (and associated documentation) is available and maintaining a program of unknown origin. This topic addresses the former, while the next topic deals with the latter. Part of the effort of maintaining an engineered system includes preserving the structure and integrity of the system.

- Life cycle model for maintenance [Boehm88, Wegner84]
- Top-down strategies for introducing change
- Preserving design integrity
- Code reading [Goldberg87]

Maintaining old code

When the original design is not present, it must be recreated from the code. This process of reverse engineering requires skills of code understanding that are developed in the Software Verification and Validation course.

- Life cycle model for maintenance [Lehman84, Lehman85]
- Bottom-up strategies for introducing change
 - Reverse engineering [Linger79, Britcher86]
 - Code restructuring
 - Code reading
 - Recording abstractions
 - Analyzing interfaces/coupling [Wilde87a]
- Creating information hiding modules
- Reducing coupling
- Bottom-up and top-down strategies for design creation

Management of software maintenance [Lientz80], [Grady87]

Maintenance management and project management differ in that they often have different objectives. However, there are some issues that are common to both, such as configuration management.

- Developing and preserving product data [Freeman87]
 - Specifications and designs
 - Change histories

- Design rationale
- User's guide
- Records of costs
- Planning release cycles, configuration management
- Making cost tradeoffs
 - Increasing complexity vs. restructuring
 - Evaluating user's cost of change vs. producer's cost of change
 - Identifying error-prone modules [Gremillion84]
 - Investing in tools [Shneiderman86]
- Quality issues [Collofello87]

This topic overlaps with Software Verification and Validation, but provides a different perspective for the purpose of testing.

 - Reviews and inspections
 - Regression testing
 - Test cases for new function
- Productivity issues [Holbrook87, Wilde87b]
- Maintenance-specific tools typically support reverse engineering.*
 - Code restructurers
 - Code analyzers [Cleveland87, Ince85]
 - Data analyzers
 - Constructors

Relevant SEI Curriculum Modules

CM-3	<i>The Software Technical Review Process</i> , James S. Collofello
CM-4	<i>Software Configuration Management</i> , James E. Tomayko
CM-7	<i>Assurance of Software Quality</i> , Bradley J. Brown
CM-10	<i>Models of Software Evolution: Life Cycle and Process</i> , Walt Scacchi
CM-12	<i>Software Metrics</i> , Everaldo E. Mills

Pedagogical Concerns

An instructor in this course should have had experience in developing and maintaining software of significant size. Assignments in this course should involve using pre-existing code at least as much as creating new code. Software maintenance assignments should involve working with a significant existing product and changing it according to specified requirements. A code artifact would be useful in this context [Engle89].

Because of the nature of code reading, software generation assignments may be small and frequent, if desired. Because of the nature of code modification, software maintenance assignments are likely to be large and may last for the full length of the maintenance portion of the course.

Comments

Although generation of new code and maintenance of old code are distinctly different activities, the skills required to analyze code are common to both. Also, it is best to discuss the consequences of implementation (maintenance) soon after describing the implementation process (code generation).

There are several competing philosophies about maintenance, how it is best characterized, and how it might best be taught. Three of these philosophies are:

- Maintenance is a unique activity requiring special skills.
- Maintenance is not intrinsically different from software development activities, but it has a different set of constraining factors (such as the existence of body of code).
- Maintenance activities should focus on the specification for the software rather than the code, with other activities being derived as in development.

Each implementation of this course is likely to be different from the others because of these philosophical differences. It is to be hoped that significant lessons can be learned from the first few implementations.

Bibliography

- | | |
|--------------|--|
| Bentley82 | Bentley, J. L. <i>Writing Efficient Programs</i> . Englewood Cliffs, N.J.: Prentice-Hall, 1982. |
| Bentley86 | Bentley, J. L. <i>Programming Pearls</i> . Reading, Mass.: Addison-Wesley, 1986. |
| Boehm88 | Boehm, B. W. "A Spiral Model of Software Development and Enhancement." <i>Computer</i> 21, 5 (May 1988), 61-72. |
| Britcher86 | Britcher, R. N., and Craig, J. J. "Using Modern Design Practices to Upgrade Aging Software Systems." <i>IEEE Software</i> 3, 3 (May 1986), 16-24. |
| Clapp81 | Clapp, J. A. "Designing Software for Maintainability." <i>Computer Design</i> 20, 9 (Sept. 1981). |
| Cleveland87 | Cleveland, L. <i>An Environment for Understanding Programs</i> . Tech. Rep. 12880, IBM, 1987. |
| Collofello87 | Collofello, J. S., and Buck, J. J. "Software Quality Assurance for Maintenance." <i>IEEE Software</i> 4, 5 (Sept. 1987), 46-51. |
| Engle89 | Engle, C. B., Jr., Ford, G., and Korson, T. <i>Software Maintenance Exercises for a Software Engineering Project Course</i> . Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989. |
| Freeman87 | Freeman, P. <i>Software Perspectives: The System is the Message</i> . Reading, Mass.: Addison-Wesley, 1987. |
| Glass81 | Glass, R., and Noiseux, R. A. <i>Software Maintenance Guidebook</i> . Englewood Cliffs, N.J.: Prentice-Hall, 1981. |

- Goldberg87 Goldberg, A. "Programmer as Reader." *IEEE Software* 4, 5 (Sept. 1987), 62-70. Reprinted from *Information Processing 86*, H. J. Kugler, ed., North-Holland, Amsterdam, 1986.
- Grady87 Grady, R. B. "Measuring and Managing Software Maintenance." *IEEE Software* 4, 5 (Sept. 1987), 35-45.
- Gremillion84 Gremillion, L. L. "Determinants of Program Repair Maintenance Requirements." *Comm. ACM* 27, 8 (Aug. 1984), 826-832.
- Holbrook87 Holbrook, H. B., and Thebaut, S. M. *A Survey of Maintenance Tools that Enhance Program Understanding*. Tech. Rep. SERC-TR-9-F, Software Engineering Research Center, Purdue Univ.-Univ. of Florida, Sept. 1987.
- Ince85 Ince, D. C. "A Program Design Language Based Software Maintenance Tool." *Software Practice and Experience* 15, 6 (June 1985).
- Lehman84 Lehman, M. M. "A Further Model of Coherent Programming Processes." *Proc. Software Process Workshop*, Colin Potts, ed. IEEE Computer Society Press, Feb. 1984, 27-34.
- Lehman85 Lehman, M. M. *Program Evolution: Processes of Software Change*. London: Academic Press, 1985.
- Lientz80 Lientz, B. P., and Swanson, E. *Software Maintenance Management*. Reading, Mass.: Addison-Wesley, 1980.
- Linger79 Linger, R. C., Mills, H. D., and Witt, B. I. *Structured Programming: Theory and Practice*. Reading, Mass.: Addison-Wesley, 1979.
- Martin83 Martin, J., and McClure, C. *Software Maintenance: The Problem and Its Solutions*. Englewood Cliffs, N.J.: Prentice-Hall, 1983.
- Parnas79 Parnas, D. L. "Designing Software for Ease of Extension and Contraction." *Trans. Soft. Eng. SE-5*, 2 (Mar. 1979).
- Shneiderman86 Shneiderman, B., Shafer, P., Simon, R., and Weldon, L. "Display Strategies for Program Browsing: Concept and Experiment." *IEEE Software* 3, 3 (May 1986), 7-15.
- Wegner84 Wegner, P. "Capital-Intensive Software Technology." *IEEE Software* 1, 3 (July 1984), 7-45.
- Wilde87a Wilde, N., and Nejme, B. *Dependency Analysis: An Aid for Software Maintenance*. Tech. Rep. SERC-TR-13-F, Software Engineering Research Center, Purdue Univ.-Univ. of Florida, Sept. 1987.
- Wilde87b Wilde, N., and Thebaut, S. M. *The Maintenance Assistant: Work in Progress*. Tech. Rep. SERC-TR-10-F, Software Engineering Research Center, Purdue Univ.-Univ. of Florida, Sept. 1987. To be published in *Journal of Systems and Software*.

3.4.5. Software Verification and Validation

Catalog Description

This course addresses the theory and practice of ensuring high-quality software products. Topics covered include quality assessment, proof of correctness, testing, and limitations of verification and validation methods.

Course Objectives

After completing this course, students should be able to prepare an effective test plan, analyze a test plan, apply systematic integration testing, prove a module correct, and plan and conduct a technical review.

Prerequisites

A second-semester course in computer science (such as data structures) and a discrete mathematics course.

Syllabus

Wks	Topics and Subtopics (<i>Objective</i>)
-----	---

0.5	Verification and Validation Limitations (<i>Knowledge</i>)
-----	--

Students should be made aware of the theoretical and practical limitations of testing and program proving. Validation is limited by the informal nature of user requirements.

Review of concepts and terminology [Goodenough75]

0.5	Definition and Assessment of Product Quality (<i>Knowledge</i>)
-----	---

Quality is difficult to define, but users claim that it is easy to recognize. One quantifiable measure is the number of errors reported. Configuration management typically tracks this kind of data, providing a relationship between this course and the Software Project Management course.

Product quality factors

Assessment of product quality

3.5	Proof of Correctness Methods (<i>Application</i>)
-----	---

This topic ensures that students are familiar with the latest methods and problems in this area. The skills they develop will help them read and analyze code for other purposes, such as maintenance.

Functional correctness [Mills86]

Weakest precondition [Dijkstra76]

Procedures [Hoare71]

Algebraic [Guttag78]

2.5 Technical Reviews (*Analysis*)

Early reviews have been the most cost-effective means of eliminating errors in software. Students should learn how to plan, conduct, and participate in several different forms of reviews (e.g., walkthroughs, inspections).

6 Testing (*Comprehension*)

Although the educational objective for this topic is comprehension, some of the subtopics should achieve higher levels. For example, students should reach the application level for some specific module-level testing methods. (They may only achieve comprehension for other methods.) It is important to cover the entire life cycle, especially those methods that apply to entire systems.

Module-level testing methods (functional, structural, error-oriented, hybrid)

Integration

Test plans and documentation

Transaction flow analysis

Stress analysis (failure, concurrency, performance)

1 Test Environments (*Comprehension*)

Students should recognize which tasks and aspects of testing are amenable to automation and which require human intervention. The goal should be to automate as many tasks as feasible.

Tools

Environments for testing

Relevant SEI Curriculum Modules

CM-3 *The Software Technical Review Process*, James S. Collofello

CM-7 *Assurance of Software Quality*, Bradley J. Brown

CM-9 *Unit Testing and Analysis*, Larry J. Morell

CM-13 *Introduction to Software Verification and Validation*, James S. Collofello

Pedagogical Concerns

It is important to convey the applicability of the methods. For example, proof of correctness is currently applicable only to modules, while testing is more suitable for systems.

Comments

It is assumed that students will have seen some proof of correctness methods in their undergraduate program. For example, weakest preconditions are often taught in an early programming course. However, most students will need to review these topics in this course.

Bibliography

- Dijkstra76 Dijkstra, E. *A Discipline of Programming*. Englewood Cliffs, N.J.: Prentice-Hall, 1976.
- Goodenough75 Goodenough, J. B., and Gerhart, S. L. "Toward a Theory of Test Data Selection." *Trans. Soft. Eng. SE-1*, 2 (June 1975).
- Guttag78 Guttag, J. V., Horowitz, E., and Musser, D. R. "Abstract Data Types and Software Validation." *Comm. ACM* 21, 12 (Dec. 1978), 1048-1064.
- Hoare71 Hoare, C. A. R. "Procedures and Parameters - An Axiomatic Approach." *Symp. Semantics of Algor. Lang.*, E. Engeler, ed. 1971.
- Mills86 Mills, H. D., Basili, V. R., Gannon, J. D., and Hamlet, R. G. *Principles of Computer Programming, A Mathematical Approach*. Allyn and Bacon, 1986.

3.4.6. Software Project Management

Catalog Description

This course addresses process considerations in software systems development. It provides advanced material in software project planning, mechanisms for monitoring and controlling projects, and leadership and team building.

Course Objectives

After completing this course, students should know how to develop a software project management plan; how to set up monitoring and controlling mechanisms for software projects; how to allocate and reallocate project resources; and how to track schedule, budget, quality, and productivity. In addition, students should understand the relationships among quality assurance, configuration management, and project documentation. They should gain an understanding of the key issues in motivating workers and leading project teams. They should be aware of intellectual property issues, software contracting and licensing, and process assessments.

Prerequisites

There are no specific prerequisites beyond admission to the MSE program.

Syllabus

Wks	Topics and Subtopics (<i>Objective</i>)
-----	---

4	Introduction (<i>Comprehension</i>)
---	---------------------------------------

Students need to see the "big picture" of software development. They also need to be motivated to study the problems of management.

Software engineering process

Process models (waterfall, incremental, spiral, rapid prototype, domain)

Organizational structures (functional, matrix, individual roles)

Motivational case studies

Problematical projects (Project Foul, Medinet, *Scientific American*, OS/360, Multics, *Soul of a New Machine*)

Successful projects (GE RC2000, NASA space shuttle, ESS #1, Olympics message system)

Huge systems (air traffic control, Strategic Defense Initiative)

Project origins

Requests for proposals (RFP), statements of work (SOW), contracts, business plans

System requirements

Software requirements

- Legal issues
 - Intellectual property rights
 - Contracts
 - Licensing
 - Liability
 - Post-employment agreements

4.5 Planning (*Application*)

Good planning is still considered an art rather than a science. However, students should learn how to use the best methods available. It is important to stress the importance of tailoring any method to the problem and the environment.

- Standards
 - External (2167A, 2168, NASA, IEEE)
 - Internal (corporate, project)
 - Tailoring
- Work breakdown
- Scheduling
 - CPM, PERT, activity networks
 - Milestones and work products
- Resources
 - Acquisition
 - Allocation
 - Tradeoffs
- Risk analysis
 - Identification
 - Assessment
 - Contingency planning
- Estimates
 - Expert judgment (individual, Delphi)
 - Size estimates
 - Models (driven by lines of code, by function point; time-sensitive models)

4.5 Monitoring and Controlling (*Application*)

Much of this topic deals with issues of product quality. There is an overlap here with material from the Software Verification and Validation course. The subtopic on leadership may be difficult to teach, but its inclusion in the course is important, if only to stimulate awareness of the different kinds of problems found in this area.

- Process metrics
 - Quality
 - Schedule
 - Budget
 - Productivity
- Earned value tracking
- Quality assurance

- Technical reviews (walkthroughs, inspections, acceptance testing)
- Planning
- Configuration management
 - Planning
 - Identification
 - Change control
 - Auditing
 - Tools
- Risk management
 - Tracking
 - Crisis management
- Leadership, training, and motivation
 - Work environment
 - Motivation and job satisfaction
 - Leadership styles
 - Team structures (hierarchical, chief programmer, democratic)
 - Productivity assessment
 - Performance reviews
 - Small group dynamics

1 Project Assessment (*Application*)

Students should assess one another's work. This is one of the best ways to synthesize material from several topics of the course. For example, the combined effects of poor planning and poor control are best seen through postmortem analysis. Students should be given the opportunity to fail, since they will be less willing to try novel approaches outside academia.

- In-process assessment
- Final assessment
- Project formation
 - Postmortems and lessons learned
 - Summary data collection
 - Staff reassignments

Relevant SEI Curriculum Modules

CM-3	<i>The Software Technical Review Process</i> , James S. Collofello
CM-4	<i>Software Configuration Management</i> , James E. Tomayko
CM-7	<i>Assurance of Software Quality</i> , Bradley J. Brown
CM-10	<i>Models of Software Evolution: Life Cycle and Process</i> , Walt Scacchi
CM-14	<i>Intellectual Property Protection for Software</i> , Pamela Samuelson and Kevin Deasy
CM-12	<i>Software Metrics</i> , Everald E. Mills
CM-21	<i>Software Project Management</i> , James E. Tomayko and Harvey K. Hallman

Pedagogical Concerns

A project should be assigned; it should primarily involve planning--no implementation need be done.

It is difficult to provide motivation for many of the topics in this course without experience managing software development projects. Guest lecturers may be especially helpful for this.

Many aspects of software maintenance may be considered project management issues. Instructors should coordinate the coverage of these topics between this course and the Software Generation and Maintenance course.

Bibliography

The bibliography for this course is still being developed. The bibliography of curriculum module CM-21 provides useful references for most of this course.

3.5. Project Experience Component

In addition to coursework covering the units described above, the curriculum should incorporate a significant software engineering experience component representing at least 30% of the student's work. Universities have tried a number of approaches to give students this experience; examples are summarized in Figure 3.1.

School	Approach	Description
Seattle University, Monmouth College, Texas Christian University	Capstone project course	Students do a software development project after completion of most coursework
University of Southern California	Continuing project	Students participate in the <i>Software Factory</i> , a project that continues from year to year, building and enhancing software engineering tools and environments
Arizona State University	Multiple course coordinated project	A single project is carried through four courses (on software analysis, design, testing, and maintenance); students may take the courses in any order
University of Stirling	Industry cooperative program	After one year of study, students spend six months in industry on a professionally managed software project, followed by a semester of project or thesis work based in part on the work experience
Imperial College	Commercial software company	Students participate in projects of a commercial software company that has been established by the college in cooperation with local companies
Carnegie Mellon University	Design studio	Students work on a project under the direction of an experienced software designer, similar to a <i>master-apprentice</i> relationship

Figure 3.1. Approaches to the experience component

One form of experience is a cooperative program with industry, which has been common in undergraduate engineering curricula for many years. The University of Stirling uses this form in their Master of Science in Software Engineering program [Budgen86]. Students enter the program in the fall semester of a four-semester program. Between the first and second semesters, they spend two or three weeks in industry to learn about that company. They return to the company in July for a six-month stay, during which time they participate in a professionally managed project. The fourth semester is devoted to a thesis or project report, based in part on their industrial experience.

Imperial College of Science and Technology has a similar industry experience as part of a four-year program leading to a Bachelor of Science in Engineering degree [Lehman86]. For this purpose, the College has set up Imperial Software Technology, Ltd. (IST) in partnership with the National Westminster Bank PLC, The Plessey Company PLC, and PA International. IST is an independent, technically and commercially successful company that provides software technology products and services.

The more common form of experience, however, is one or more project courses as part of the curriculum. Two forms are common: a project course as a capstone following all the lecture courses, and a project that is integrated with one or more of the lecture courses.

The Wang Institute of Graduate Studies (before it closed in 1987), Texas Christian University, and Seattle University have each offered a graduate software engineering degree for several years, and the College of St. Thomas is in its fifth year of offering its degree program. Each school incorporates a capstone project course into its curriculum. The Wang Institute often chose projects related to software tools that could be useful to future students. TCU takes the professional backgrounds of its students into consideration when choosing projects. Seattle sometimes solicits real projects from outside the university. The College of St. Thomas allows students to work on projects for their employers, other than their normal work assignments.

It is worth noting that the project course descriptions for all four of these institutions do not mention software maintenance. Educators and practitioners alike have long recognized that maintenance requires the majority of resources in most large software systems. The lack of coverage of maintenance in software engineering curricula may be attributed to several factors. First, there does not appear to be a coherent, teachable body of knowledge on software maintenance. Second, current thinking on improving the maintenance process is primarily based on improving the development process; this includes the capturing of development information for maintenance purposes. Finally, giving students maintenance experience requires that there already exists a significant software system with appropriate documentation and change requests, the preparation of which requires more time and effort than an individual instructor can devote to course preparation. (The SEI has published some materials to address this final problem [Engle89].)

The University of Southern California has built an infrastructure for student projects that continue beyond the boundaries of semesters and groups of students. The *System Factory Project* [Scacchi86] has created an experimental organizational environment for developing large software systems that allows students to encounter many of the problems associated with professional software engineering and to begin to find effective solutions to the problems. To date, more than 250 graduate students have worked on the project and have developed a large collection of software tools.

The University of Toronto has added the element of software economics to its project course [Horning76, Wortman86]. The *Software Hut* (a small software house) approach requires student teams to build modules of a larger system, to try to sell their module to other teams (in competition with teams that have developed the same module), to evaluate and buy other modules to complete the system, and to make changes in purchased modules. At the end of the course, systems are "sold" to a "customer" at prices based on the system quality (as

determined by the instructor's letter grade for the system). The instructor reports that this course has a very different character from previous project courses. The students' attempts to maximize their profits gave the course the flavor of a game and helped motivate students to use many techniques for increasing software quality.

Arizona State University has built the project experience into a sequence of courses, combining lectures with practice [Collofello82]. The four courses were Software Analysis (requirements and specifications), Software Design, Software Testing, and Software Maintenance. The courses were offered in sequence so that a single project could be continued through all four. However, the students could take the courses in any order, and although many students did take them in the normal (waterfall model) order, the turnover in enrollment from one semester to the next gave a realistic experience.

Carnegie Mellon University has recently initiated an MSE degree program based in part on the SEI curriculum described in this report. This program is experimenting with a year-long *design studio* approach to the project experience component, in which students work closely with faculty on software development; this is similar to the *master-apprentice* model.

We do not believe that there is only one *correct* way to provide software engineering experience. It can be argued that experience is the basis for understanding the abstractions of processes that make up formal methods and that allow reasoning about processes. Therefore, we should give the students experience first, with some guidance, and then show them that the formalisms are abstractions of what they have been doing. It can also be argued that we should teach "theory" and formalisms first, and then let the students try them in capstone project courses.

No matter what form the experience component takes, it should provide as broad an experience as possible. It is especially important for the students to experience, if not perform, the control activities and the management activities (as defined in Appendix 1). Without these, the project can be little more than advanced programming.

3.6. Electives

Electives may make up 20% to 40% of a curriculum. Although it is a young discipline, software engineering is already sufficiently broad that students can choose specializations (such as project management, systems engineering, or real-time systems); there is no "one size fits all" MSE curriculum. The electives provide the opportunity for that specialization.

In addition, there is a rather strong perception among industrial software engineers that domain knowledge for their particular industry is essential to the development of effective software systems. Therefore, we also suggest that an MSE curriculum permit students to choose electives from the advanced courses in various application domains. Software engineers with a basic knowledge of avionics, radar systems, or robotics, for example, are likely to be in great demand. Furthermore, there is increasing evidence that better software project management can significantly influence the cost of software, so electives in management topics may be appropriate.

To summarize, there are five recommended categories of electives:

1. Software engineering subjects, such as software development environments
2. Computer science topics, such as database systems or expert systems
3. Systems engineering topics, especially topics at the boundary between hardware and software
4. Application domain topics
5. Engineering management topics

3.7. Pedagogical Considerations

Software engineering is difficult to teach for a variety of reasons. It is a relatively new and rapidly changing discipline, and it has aspects of an art and a craft as well as a science and an engineering discipline. As a result, educators must develop a variety of teaching techniques and materials in order to provide effective education.

Psychologists distinguish *declarative* knowledge and *procedural* knowledge [Norman88]. The former is easy to write down and easy to teach; the latter is nearly impossible to write down and difficult to teach. It is largely subconscious, and it is best taught by demonstration and best learned through practice. Many of the processes of software engineering depend on procedural knowledge. It is for this reason that we recommend such a significant amount of project experience (see Section 3.5).

Another aspect of experience that can be built into the curriculum involves "tricks of the trade." Software engineers, during the informal apprenticeship of their first several years in the profession, are likely to be exposed to a large number of recurring problems for which there are accepted solutions. These problems and solutions will vary considerably from one application domain to another, but all software engineers seem to accumulate them in their "bags of tricks."

We believe that students would receive some of the benefits of their "apprenticeship" period while still in school if these problems and solutions were included in the curriculum. For this reason, we have included large course segments titled "Paradigms" in the specification and design courses (see the descriptions of these courses following this report).

The principal definition of the word *paradigm* is "EXAMPLE, PATTERN; *esp* : an outstandingly clear or typical example or archetype" [Webster83]. The word *archetype* is defined in the same source as "the original pattern or model of which all things of the same type are representations or copies : PROTOTYPE; *also* : a perfect example." We believe that these definitions capture the notion of a widely accepted or demonstrably superior solution to a recurring problem.

Unfortunately, there is no ready source of appropriate paradigms. The paradigms sections of the specification and design courses only hint at the kinds of material to be presented. Therefore the SEI Education Program has begun efforts to identify and document paradigms

in a number of software application domains. We hope to report initial success in this endeavor in our next curriculum report.

3.8. The Structure of the MSE Curriculum

A typical master's degree curriculum requires 30 to 36 semester hours[†] credit. The courses described in Section 3.4 require three hours each, totaling 18 semester hours. This allows time for the project experience component and for some electives.

Because of the wide range of choices for electives, students can be well served by creative course design. For example, several small units of material (roughly one semester hour each) might be prepared by several different instructors. Three of these could then be offered sequentially in one semester under the title "Topics in Software Engineering," with different units offered in different semesters.

Figure 3.2 shows the structure of a curriculum based on the six core courses. This structure reflects the familiar *spiral approach* to education, in which material is presented several times in increasing depth. This approach is essential for a discipline such as software engineering, with many complex interrelationships among topics; no simple linear ordering of the material is possible.

Students learn the basics of computer science and programming-in-the-small in the undergraduate curriculum. The six core courses build on these basics by adding depth, formal methods, and the programming-in-the-large concepts associated with systems engineering and the control and management activities. The electives and the project experience component provide further depth and an opportunity for specialization.

[†]Note for readers not familiar with United States universities: A *semester hour* represents one contact hour (usually lecture) and two to three hours of outside work by the student per week for a semester of about fifteen weeks. A *course* covers a single subject area of a discipline, and typically meets three hours per week, for which the student earns three semester hours of credit. A graduate student with teaching or research responsibilities might take three courses (nine semester hours) each semester; a student without such duties might take five courses.

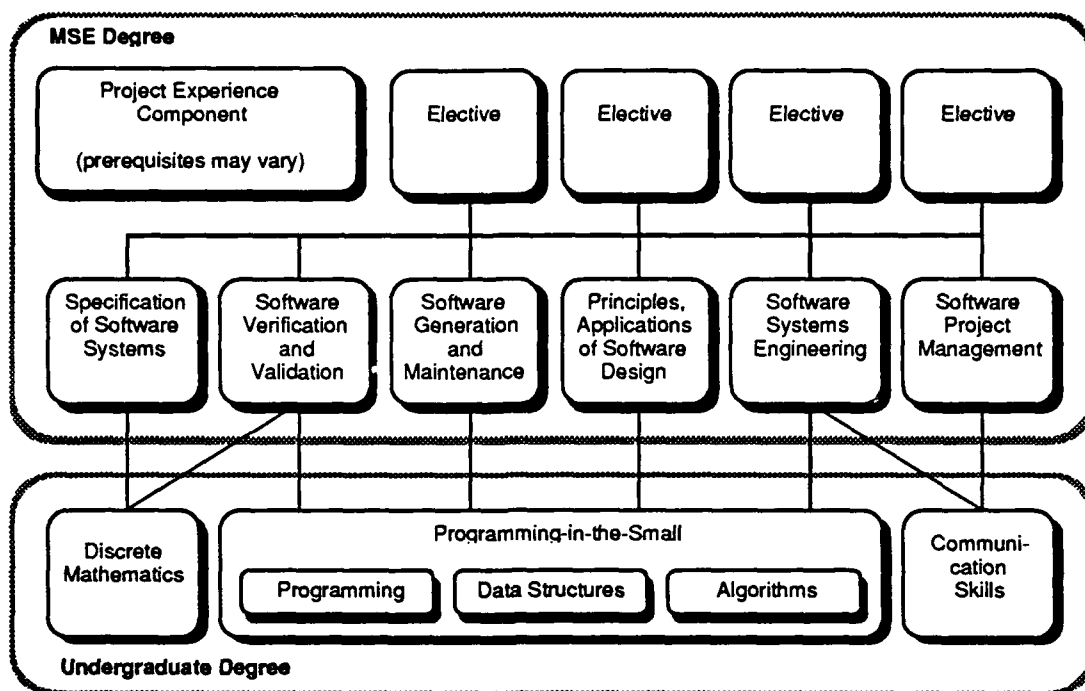


Figure 3.2. MSE curriculum structure

4. Survey of Graduate Degree Programs in Software Engineering

Graduate degree programs first appeared in the late 1970s at Texas Christian University, Seattle University, and the Wang Institute of Graduate Studies. All three programs responded to significant needs from local industry in the Dallas/Fort Worth, Seattle, and Boston areas, respectively. In 1985, three additional programs were started: at the College of St. Thomas in St. Paul, Minnesota, at Imperial College of Science and Technology in London, and at the University of Stirling in Scotland. The last four years have seen a significant increase in the development of and interest in such programs. We know of at least a dozen programs that either have been initiated or are under development.

In this section, we survey the programs in the United States and Europe for which we were able to obtain information. Readers will note substantial variation among the programs. This can be attributed to a number of factors:

- Most of the programs were developed in the absence of any recognized model curriculum.
- Each school had a number of existing courses, mostly in computer science, that were incorporated into the new programs, and these courses differed greatly among schools.
- Software engineering is a new discipline, and the developers of these programs had differing perceptions of the scope of the discipline, and its principles and practices.
- Each school was responding to perceived needs that varied greatly from one community to another.

Another notable point of variation among these programs is the program title (see Figure 4.1). Many of the programs were unable to use the word *engineering* in their titles because of legal or administrative restrictions. In one way, it is unfortunate that the term *software engineering* is so nearly universally accepted as an informal name for the discipline, because it has caused an inordinate amount of time and energy to be devoted to arguing semantic issues of whether software engineering is really engineering.

We believe it is valuable for a school considering the development of a graduate program in software engineering to examine not only the SEI recommendations but also these existing programs. Therefore we have sketched the requirements for each program below.

Program Title	University
Master of Software Engineering	Carnegie Mellon University Seattle University Wang Institute of Graduate Studies (former)
Master of Science in Software Engineering	Andrews University Monmouth College University of Houston-Clear Lake (proposed) University of Stirling The Wichita State University
Master of Computer Science in Software Engineering	The Wichita State University
Master of Science in Software Systems Engineering	Boston University George Mason University
Master of Software Design and Development	College of St. Thomas Texas Christian University
Master of Science in Software Development and Management	Rochester Institute of Technology
Master of Engineering	Imperial College of Science and Technology
Software Engineering Curriculum Master	Polytechnic University of Madrid

Figure 4.1. Software engineering degree program titles

Andrews University

<i>Location</i>	Berrien Springs Michigan
<i>Program title</i>	Master of Science in Software Engineering
<i>Degree requirements</i>	48 quarter credits (typically 4 credits per course): 8 credits of projects, 16 credits core courses, 0-20 credits foundation courses, 4-24 credits electives.
<i>Foundation courses</i>	Data Structures Data Base Systems Systems Analysis I Systems Analysis II Operating Systems
<i>Core courses</i>	Computer Architecture Software Engineering I Software Engineering II Programming Project Management
<i>Program initiation</i>	(unknown)
<i>Source</i>	This information was reported to the SEI by Andrews University in April 1989.

Boston University

<i>Location</i>	Boston, Massachusetts
<i>Program title</i>	Master of Science in Software Systems Engineering
<i>Degree requirements</i>	Nine courses of four credits each: seven required courses including a project course, and two electives. Two of the required courses differ depending on whether the student's background is in hardware or software.
<i>Required courses</i>	Applications of Formal Methods Software Project Management Software System Design Computer as System Component Software Engineering Project Advanced Data Structures (<i>hardware background</i>) Operating Systems (<i>hardware background</i>) Switching Theory and Logic Design (<i>software background</i>) Computer Architecture (<i>software background</i>)
<i>Program initiation</i>	Fall 1989 (The program has existed as a software engineering option in the Master of Science in Systems Engineering since spring 1980; the current curriculum was adopted in January 1988.)
<i>Source</i>	This information was taken from [Brackett88].

Boston University absorbed the Wang Institute's facilities in 1987 and was the beneficiary of some of the experience of the Wang Institute. This program incorporates the best features of the MSE curriculum of Wang and the MS in Systems Engineering from Boston University. The program emphasizes the understanding of both hardware and software issues in the design and implementation of software systems. Special emphasis is placed on the software engineering of two important classes of computer systems: embedded systems and networked systems.

Both full-time and part-time programs are available, and most of the program is available through the Boston University Corporate Classroom interactive television system. The program can be completed in twelve months by full-time students.

The university also has a doctoral program leading to the PhD in Engineering, with research specialization in software engineering.

Carnegie Mellon University

<i>Location</i>	Pittsburgh, Pennsylvania
<i>Program title</i>	Master of Software Engineering
<i>Degree requirements</i>	<i>(This information is tentative.)</i> Sixteen courses: six required courses and two Category I electives in the first year; a theory course, a business course, two Category II electives, two software engineering seminars, and a two-semester master's project in the second year.
<i>Required courses</i>	Software Systems Engineering Formal Methods in Software Engineering Advanced System Design Principles Software Creation and Maintenance Analysis of Software Software Project Management
<i>Electives</i>	Category I: computer science courses at the senior undergraduate level Category II: advanced graduate courses in computer science
<i>Prerequisite note</i>	Prospective students must have at least two years of experience working in a sizable software project.
<i>Program initiation</i>	September 1989
<i>Source</i>	This information was reported to the SEI by CMU in June 1989.

The objective of Carnegie Mellon University's MSE program is to produce a small number of highly skilled experts in software system development. It is designed to elevate the expertise of practicing professional software designers. The emphasis is on practical application of technical results from computer science; the nature of these technical results dictates a rigorous, often formal, orientation. The engineering setting requires responsiveness to the needs of end users in a variety of application settings, so the program will cover resolution of conflicting requirements, careful analysis of tradeoffs, and evaluation of the resulting products. Since most software is now produced by teams in a competitive setting, the program will also cover project organization, scheduling and estimation, and the legal and economic issues of software products.

College of St. Thomas

<i>Location</i>	St. Paul, Minnesota
<i>Program title</i>	Master of Software Design and Development
<i>Degree requirements</i>	Ten required courses, including a two-semester project course sequence, and four elective courses. All courses are three semester credits.
<i>Required courses</i>	Technical Communications Programming Methodologies DBMS and Design Systems Analysis and Design I Software Productivity Tools I Software Project Management Software Quality Assurance/Quality Control Legal Issues in Technology
<i>Program initiation</i>	February 1985
<i>Source</i>	This information was reported to the SEI by the College of St. Thomas in June 1989.

This program was developed through an advisory committee made up of technical managers from Twin Cities companies such as Honeywell, IBM, Sperry, 3M, NCR-Comten, and Control Data. Elective courses are added to the curriculum on the basis of need as expressed by technical managers in local industry or by students in the program.

The program is applied rather than research-oriented. Most instructors are from industry (14 of 23 in the spring 1989 semester). Instead of a thesis, students complete a two semester software project in a local company; in many cases this company is their employer, but the project must not be part of their normal work responsibilities.

Classes are offered evenings, and 98% of students work full-time in addition to their studies. Students normally require three years to complete the degree. The program enrolled 252 students in spring 1989.

George Mason University

<i>Location</i>	Fairfax, Virginia
<i>Program title</i>	Master of Science in Software Systems Engineering
<i>Degree requirements</i>	30 hours of course work in the School of Information Technology and Engineering, including five required courses.
<i>Required courses</i>	Introduction to Software Engineering Formal Methods in Software Engineering Software Requirements, Prototyping, and Design Software Project Management Software Project Laboratory
<i>Electives</i>	Five courses, including a second semester of Software Project Laboratory, or three courses and 6 semester hours of master's thesis.
<i>Program initiation</i>	Fall 1989 (core courses offered beginning Fall 1988)
<i>Source</i>	This information was reported to the SEI by George Mason University in April 1989.

The program for the degree of Master of Science in Software Systems Engineering is concerned with engineering technology for developing and modifying software components in systems that incorporate digital computers. The program is concerned with both technical and managerial issues, but primary emphasis is placed on the technical aspects of building and modifying software systems.

In addition to the degree program, the university offers a graduate certificate program in software systems engineering. The program is designed to provide knowledge, tools, and techniques to those who are working in, or plan to work in, the field of software systems engineering, but do not desire to complete all of the requirements for a master's degree. Students in the certificate program must already hold or be pursuing a master's degree in a science or engineering discipline. The requirements for the certificate are completion of the five required courses listed above.

Imperial College of Science and Technology

<i>Location</i>	London, England
<i>Program title</i>	Master of Engineering
<i>University structure</i>	British universities normally have three-year bachelor's degree programs; the master of engineering is a four-year first degree program. In its first two years the program is the same as the (three-year) bachelor of science program in computer science.
<i>Degree requirements</i>	Third and fourth year coursework includes compulsory courses totaling three modules and optional courses totaling six modules (each module represents 22 hours of lecture). During the third year, students spend approximately six months in industry; during the fourth year they must complete an individual project.
<i>Compulsory courses</i>	<p>(these courses total six modules)</p> <p>Software Engineering Process Calculus of Software Development Database Technology Introduction to Macro Economics and Financial Management Introduction to Management Methodology of Software Development Language Definition and Design Programming Support Environments Standards, Ethical and Legal Considerations</p>
<i>Optional courses (third year)</i>	<p>(one module each)</p> <p>Functional Programming Technology I Artificial Intelligence Technology Compiler Technology Computer Networks Object Oriented Architecture Interface and Microprocessor Technology Performance Analysis of Computer Systems Graphics Silicon Compilation Applied Mathematics Industrial Sociology Government Law and Industry Humanities</p>

<i>Optional courses (fourth year)</i>	<i>(one module each)</i> Advanced Logic Theorem Proving Concurrent Computation Human-Computer Interaction Expert Systems Technology Functional Programming Technology II Advanced Operation Systems Parallel Architecture Distributed Systems VLSI Robotics Computing in Engineering Natural Language Processing Micro-Economic Concepts Industrial Relations Innovation and Technical Change Humanities
<i>Program initiation</i>	Fall 1985
<i>Source</i>	This information was taken from [Lehman86].

Since British students normally must commit to either a three-year (bachelor's degree) or a four-year (master's degree) program at the end of secondary school (the student cannot complete the bachelor's degree and then decide to continue for the master's), the latter programs tend to attract the better students. Entrance requirements are generally more stringent for the master's programs and the graduates are expected to advance rapidly once they enter industry.

The industry component of this program has been described earlier in this report (Section 3.5). This component is perceived to be somewhat analogous to the role of teaching hospitals in the education of medical students.

Monmouth College

<i>Location</i>	West Long Branch, New Jersey
<i>Program title</i>	Master of Science in Software Engineering
<i>Degree requirements</i>	30 credit hours, consisting of 6 core and 4 elective courses.
<i>Core courses</i>	Mathematical Foundations of Software Engineering I Programming-in-the-Large Project Management Computer Networks Software Engineering I System Project Implementation (Laboratory Practicum)
<i>Elective courses</i>	Mathematical Foundations of Computer Science II Programming-in-the-Small Protocol Engineering Selected Topics in Software Engineering Programming Languages Computer Architecture Operating System Implementation Database Management (additional electives are under development)
<i>Program initiation</i>	1986
<i>Source</i>	This information was taken from [Amoroso88] and from information reported to the SEI by Monmouth College in April 1989.

The program is offered through the departments of computer science and electrical engineering. The current enrollment is more than 100, and to date 50 students have completed the degree requirements.

Polytechnic University of Madrid

<i>Location</i>	Madrid, Spain
<i>Program title</i>	Software Engineering Curriculum Master
<i>University structure</i>	The Spanish university system organizes its programs differently from United States universities, so this program cannot be described in terms of courses. For each of the subject areas described below, the amount of time devoted to the area is given in units. Each unit represents a 75 minute class meeting. The program totals approximately 500 units.
<i>Degree requirements</i>	<p>Introduction to Software Engineering (3)</p> <p>Models of Computation (76)</p> <p>Computing Machinery (6)</p> <p>Software Production Technology and Methodology</p> <p> Information Systems</p> <p> Introduction to Requirements Analysis (15)</p> <p> Formal Specification Techniques (25)</p> <p> Design (55)</p> <p> Implementation (85)</p> <p> Tools Evaluation (2)</p> <p> Software Engineering and Artificial Intelligence (11)</p> <p>Product and Process Control</p> <p> System Construction Management (20)</p> <p> Quality Control</p> <p> Project Management (20)</p> <p> Documentation Process (25)</p> <p>Software Product (8)</p> <p>Information Protection (14)</p> <p>Software Safety (8)</p> <p>Legal Aspects (6)</p> <p>Case Study (12)</p>
<i>Program initiation</i>	1988
<i>Source</i>	This information was reported to the SEI by Polytechnic University in May 1989.

The Polytechnic University of Madrid is the largest (well over 100,000 students) and most prestigious of the Spanish technical universities. It has large, well-established schools of engineering and informatics (computer science). The university is an academic affiliate of the SEI and has incorporated a number of SEI recommendations into its initial curriculum.

Rochester Institute of Technology

<i>Location</i>	Rochester, New York
<i>Program title</i>	Master of Science in Software Development and Management
<i>Degree requirements</i>	48 credits (quarter system; typical course is 4 credits)
<i>Required courses</i>	Principles of Software Design Principles of Distributed Systems Principles of Data Management Software and System Engineering Project Management Organizational Behavior Analysis and Design Techniques, or Analysis & Design of Embedded Systems Software Verification and Validation Software Project Management Technology Management Software Tools Laboratory Software Engineering Project
<i>Program initiation</i>	Fall 1987
<i>Source</i>	This information was reported to the SEI by RIT in April 1989.

The program has approximately 100 students at the RIT campus and 15 students at Griffiss Air Force Base in Rome, New York. Approximately 90% of the students attend part-time.

Seattle University

<i>Location</i>	Seattle, Washington
<i>Program title</i>	Master of Software Engineering
<i>Degree requirements</i>	45 credits (quarter system), including eight required core courses, four elective courses, and a three quarter project sequence.
<i>Required courses</i>	Technical Communication Software Systems Analysis System Design Methodology Programming Methodology Software Quality Assurance Software Metrics Software Project Management Formal Methods
<i>Elective courses</i>	System Procurement Contract Acquisition and Administration Database Systems Distributed Computing Artificial Intelligence Human Factors in Computing Data Security and Privacy Computer Graphics Real Time Systems Organization Behavior Organization Structure and Theory Decision Theory (other electives may be selected from the MBA program)
<i>Prerequisite note</i>	Prospective students must have at least two years of professional software experience.
<i>Program initiation</i>	1978
<i>Source</i>	This information was taken from [Mills86].

Seattle University is an independent urban university committed to the concept of providing rigorous professional educational programs within a sound liberal arts background. In 1977 the university initiated a series of discussions with representatives from local business and industry, during which software engineering emerged as a critical area of need for specialized educational programs. Leading software professionals were invited to assist in the development of such a program, which was initiated the following year.

Normally, classes are held in the evenings and students are employed full-time in addition to their studies. The first students in the program graduated in 1982.

Texas Christian University

<i>Location</i>	Fort Worth, Texas
<i>Program title</i>	Master of Software Design and Development
<i>Degree requirements</i>	36 semester hours, including nine required courses and three electives; submission of a technical paper to a journal for publication.
<i>Required courses</i>	Introduction to Software Design and Development Modern Software Requirements and Design Techniques Applied Design, Programming, and Testing Techniques Management of Software Development Economics of Software Development Computer Systems Architecture Database and Information Management Systems Software Implementation Project I Software Implementation Project II
<i>Program initiation</i>	Fall 1978
<i>Source</i>	This information was taken from [Comer86].

The university established a graduate degree program in software engineering in 1978. Due to external pressure, prompted by the absence of an engineering college at TCU, the program was given its current name in 1980.

The program offers most of its courses in the evening, and all 50 students in the program are employed full-time in the Dallas/Fort Worth area.

University of Houston-Clear Lake

<i>Location</i>	Houston, Texas
<i>Program title</i>	Master of Science in Software Engineering
<i>Degree requirements</i>	36 credit hours, including 30 hours of required courses and 6 hours of electives.
<i>Required courses</i>	Specification of Software Systems Principles and Applications of Software Design Software Generation and Maintenance Software Validation and Verification Software Project Management Master's Thesis Research Advanced Operating Systems Theory of Information and Coding Synthesis of Computer Networks
<i>Elective courses</i>	Must be chosen from courses in software engineering, computer science, compute systems design, or mathematical sciences.
<i>Program initiation</i>	awaiting approval
<i>Source</i>	This information was reported to the SEI by the University of Houston-Clear Lake in March 1989.

The university has submitted a proposal to the Texas Coordinating Board for Higher Education to offer the MSSE degree; it has not yet been approved.

Five of the required courses in this degree program are based on the SEI recommendations in this report.

University of Stirling

<i>Location</i>	Stirling, Scotland
<i>Program title</i>	Master of Science in Software Engineering
<i>Degree requirements</i>	Semester 1 (September-December) Programming Methods Language Concepts Introduction to Software Engineering Computing Science Structures and Techniques Initial industrial placement visits (January) Semester 2 (February-July) Methods for Formal Specification Concurrency (half semester) Databases (half semester) Networks and Communications Elective: Expert Systems or Language Implementation Industrial project (July-December) Dissertation (January-March)
<i>Program initiation</i>	1985
<i>Source</i>	This information was reported to the SEI by the University of Stirling in April 1989.

The MSc in Software Engineering is a "specialist conversion course" intended to train graduates with a scientific background in the methods of software engineering. The students spend twelve months at the University of Stirling and six months at an industrial research and development center. Through this approach students are given an understanding of both the current engineering technology and its application in an industrial context.

The six-month placement in industry enables each candidate to participate in a project and be responsible for a particular investigation. Where practical, this may form the basis of the individual project that is undertaken during a final three-month period and then written up in the dissertation.

Wang Institute of Graduate Studies

<i>Location</i>	Tyngsboro, Massachusetts
<i>Program title</i>	Master of Software Engineering
<i>Degree requirements</i>	Eleven courses: eight required courses including two project courses, and three elective courses.
<i>Required courses</i>	Formal Methods Programming Methods Management Concepts Computing Systems Architecture or Operating Systems Software Project Management Software Engineering Methods Project I Project II
<i>Elective courses</i>	Database Management Systems User Interface Design, Implementation and Evaluation Survey of Programming Languages Expert System Technology Translator Implementation Computing Systems Architecture Operating Systems Principles of Computer Networks Programming Environments
<i>Prerequisite notes</i>	Admission requirements included at least one year of full-time software development work experience. Also required was submission of a three to four page essay on a software development or maintenance project in which the applicant had participated, an expository survey of a technical subject, or a report on a particular software tool or method.
<i>Program initiation</i>	1979
<i>Source</i>	This information was taken from [Wang86].

The Wang Institute of Graduate Studies closed in the summer of 1987. Its facilities were donated to Boston University, and its last few students were permitted to complete their degrees at BU. During its existence, the Wang program was generally considered to be the premier program of its kind. Schools considering development of an MSE program would be well advised to examine the Wang program as a model.

Wang Institute was also a pioneer in the development of a very high quality faculty with renewable fixed-term contracts rather than a tenure system. For a rapidly evolving discipline such as software engineering, where the faculty's professional experience may be at least as valuable as its academic credentials, this model for faculty evaluation and retention may be worthy of consideration by other schools as well.

The Wichita State University

<i>Location</i>	Wichita, Kansas
<i>Program title</i>	Master of Science in Software Engineering; Master of Computer Science in Software Engineering
<i>Degree requirements</i>	30 credit hours total: two required courses, six credit hours of software engineering electives, additional electives in software engineering or computer science, and practicum (3 hours) or thesis (6 hours) on a software engineering topic.
<i>Required courses</i>	Software Requirements, Specification and Design Software Testing and Validation
<i>Elective courses</i>	Software Project Management Ada and Software Engineering Systems Analysis Topics in Software Engineering (<i>recent offerings have included Configuration Management, Formal Methods, Quality Assurance, Software Metrics, and Formal Verification of Software</i>)
<i>Program initiation</i>	Spring 1989
<i>Source</i>	This information was reported to the SEI by Wichita State in June 1989.

The Wichita State University Department of Computer Science has created a set of courses than can lead to a specialization in software engineering within the existing Master of Science and Master of Computer Science degree programs. These courses are taught in cooperation with the Software Engineering Institute's Software Engineering Curriculum Project and Video Dissemination Project.

5. SEI Graduate Curriculum Test Sites

Readers of the core course descriptions in this report undoubtedly found sections in which the ordering of topics or the emphasis seemed wrong. All of the participants in the 1988 SEI Curriculum Design Workshop (where those courses were designed) expressed similar opinions. The final course descriptions incorporated a number of compromises on the ordering of topics and on the division of topics between courses. The descriptions also include several "place holder" topic headings that require further work to identify the appropriate content. We believe that continued development of these courses will be most effective if it is based on the experience gained by teaching them.

To help educators gain this experience, the SEI has established a program by which universities and other educational organizations are designated *graduate curriculum test sites*. These schools receive substantial help from the SEI in developing both courses and degree programs in software engineering. In return, the schools agree to structure their courses and programs according to SEI recommendations (to the extent appropriate for the individual school), to provide detailed reports on the level of success they achieve, and to share their teaching materials with the SEI.

The Wichita State University was designated a graduate curriculum test site in 1986. In 1989, they received state approval for a graduate curriculum in software engineering (see Section 4 of this report). They have adopted a particularly innovative and helpful approach by offering several SEI curriculum modules under the course title *Topics in Software Engineering*. This permits rapid incorporation of new material into the curriculum.

East Tennessee State University became a test site in 1989. They will pursue the establishment of an MSE degree program based on the core courses offered by the SEI Video Dissemination Project, which in turn are based on the recommendations in this report.

Carnegie Mellon University is currently developing an MSE degree program within its School of Computer Science. It is expected that some members of the SEI Education Program staff will have teaching appointments in that program. This will allow for almost immediate testing of course designs and teaching materials.

Additional graduate curriculum test sites are needed. Schools with a significant interest in the development of a graduate degree program in software engineering are invited to contact the SEI Director of Education for more information.

6. Summary and a Look Ahead

In this report, we have described our recent activities in the development of a model curriculum for a graduate professional degree in software engineering; foremost among these was the 1988 SEI Curriculum Design Workshop. The report has provided descriptions of six core courses for an MSE curriculum and a less detailed discussion of the overall curriculum, including prerequisites, electives, and project experience. The report has also surveyed 15 university graduate degree programs in software engineering.

In the coming months, the SEI Education Program will begin addressing undergraduate software engineering education. This will include sponsoring the SEI Workshop on an Undergraduate Software Engineering Curriculum. A report of our efforts is scheduled for release late in 1989. Those interested in this area should see the preliminary report on undergraduate software engineering curricula released by the British Computer Society and the Institution of Electrical Engineers [BCS89].

Software engineering continues to evolve rapidly, and software engineering education must keep pace. In the coming year we plan to identify some of the paradigms of software engineering and incorporate them into the model curriculum. We also hope to expand our efforts to help universities establish software engineering degree programs. Our progress in all these areas will be described in our next report, scheduled for release in spring 1990.

Appendix 1. An Organizational Structure for Curriculum Content

The body of knowledge called *software engineering* consists of a large number of interrelated topics. We thought it impractical to attempt to capture this knowledge as an undifferentiated mass, so an organizational structure was needed. The structure described below *is not intended to be a taxonomy of software engineering*. Rather, it is a guide that helps the SEI to collect and document software engineering knowledge and practice, and to describe the content of some recommended courses for a graduate curriculum.

Discussions of software engineering frequently describe the discipline in terms of a software life cycle: requirements analysis, specification, design, implementation, testing, and maintenance. Although these life cycle phases are worthy of presentation in a curriculum, we found this one-dimensional structure inadequate for organizing all the topics in software engineering and for describing the curriculum.

A good course, whether a semester course in a university or a one-day training course in industry, must have a central thread or idea around which the presentation is focused. Not every course can or should focus on one life cycle phase. In an engineering course (including software engineering), we can look at either the engineering *process* or the *product* that is the result of the process. Therefore, we have chosen these two views as the highest level partition of the curriculum content. Each is elaborated below.

The Process View

The process of software engineering includes several *activities* that are performed by software engineers. The range of activities is broad, but there are many *aspects* of each activity that are similar across that range. Thus, we organize those topics whose central thread is the process in two dimensions: activity and aspect.

The Activity Dimension

Activities are divided into four groups: *development*, *control*, *management*, and *operations*. Each is defined and discussed below.

Development activities are those that create or produce the artifacts of a software system. These include requirements analysis, specification, design, implementation, and testing. Because a software system is usually part of a larger system, we sometimes distinguish system activities from software activities; for example, *system design* from *software design*. We expect that many large projects will include both systems engineers and software engineers, but an appreciation of the systems aspects of the project is important for software engineers, and it should be included in a curriculum.

Control activities are those that exercise restraining, constraining, or directing influence over software development. These activities are more concerned with controlling the way in

which the development activities are performed than with producing artifacts. Two major kinds of control activities are those related to software evolution and those related to software quality.

A software product evolves in the sense that it exists in many different forms as it moves through its life cycle, from initial concept, through development and use, to eventual retirement. Change control and configuration management are activities related to evolution. We also consider software maintenance to be in this category, rather than as a separate development activity, because the difference between development and maintenance is not in the activities performed (both involve requirements analysis, specification, design, implementation, and testing), but in the way those activities are constrained and controlled. For example, the fundamental constraint in software maintenance is the pre-existence of a software system coupled with the belief that it is more cost-effective to modify that system than to build an entirely new one.

Software quality activities include quality assurance, test and evaluation, and independent verification and validation. These activities, in turn, incorporate such tasks as software technical reviews and performance evaluation.

Management activities are those involving executive, administrative, and supervisory direction of a software project, including technical activities that support the executive decision process. Typical management activities are project planning (schedules, establishment of milestones), resource allocation (staffing, budget), development team organization, cost estimation, and handling legal concerns (contracting, licensing). This is an appropriate part of a software engineering curriculum for several reasons: there is a body of knowledge about managing software projects that is different from that about managing other kinds of projects, many software engineers are likely to assume software management positions at some point in their careers, and knowledge of this material by all software engineers improves their ability to work together as a team on large projects.

Operations activities are those related to the use of a software system by an organization. These include training personnel to use the system, planning for the delivery and installation of the system, transition from the old (manual or automated) system to the new, operation of the software, and retirement of the system. Although software engineers may not have primary responsibility for any of these activities, they are often participants on teams that perform these activities. Moreover, an awareness of these activities will often affect the choices they make during the development of a software system.

The operation of software engineering support tools provides a case of special interest. These tools are software systems, and the users are the software engineers themselves. Operations activities for these systems can be observed and experienced directly. An awareness of the issues related to the use of software tools can help software engineers not only develop systems for others but also adopt and use new tools for their own activities.

The Aspect Dimension

Engineering activities traditionally have been partitioned into two categories: *analytic* and *synthetic*. We have chosen instead to consider an axis orthogonal to activities that captures

some of this kind of distinction, but that recognizes six *aspects* of these activities: *abstractions, representations, methods, tools, assessment, and communication.*

Abstractions include fundamental principles and formal models. For example, software development process models (waterfall, iterative enhancement, etc.) are models of software evolution. Finite state machines and Petri nets are models of sequential and concurrent computation, respectively. COCOMO is a software cost estimation model. Modularity and information hiding are principles of software design.

Representations include notations and languages. The Ada programming language thus fits into the organization as an implementation language, while decision tables and data flow diagrams are design notations. PERT charts are a notation useful for planning projects.

Methods include formal methods, current practices, and methodologies. Proofs of correctness are examples of formal methods for verification. Object-oriented design is a design method, and structured programming can be considered a current practice of implementation.

Tools include individual software tools as well as integrated tool sets (and, implicitly, the hardware systems on which they run). Examples are general-purpose tools (such as electronic mail and word processing), tools related to design and implementation (such as compilers and syntax-directed editors), and project management tools. Other types of software support for process activities are also included; these are sometimes described by such terms as *infrastructure, scaffolding, or harnesses.*

Sometimes the term *environment* is used to describe a set of tools, but we prefer to reserve this term to mean a collection of related representations, tools, methods, and objects. Software objects are abstract, so we can only manipulate representations of them. Tools to perform manipulations are usually designed to help automate a particular method or way of accomplishing a task. Typical tasks involve many objects (code modules, requirements specification, test data sets, etc.), so those objects must be available to the tools. Thus, we believe all four—representations, tools, methods, and objects—are necessary for an environment.

Assessment aspects include measurement, analysis, and evaluation of both software products and software processes, and of the impact of software on organizations. Metrics and standards are also placed in this category. This is an area we believe should be emphasized in the curriculum. Software engineers, like engineers in more traditional fields, need to know what to measure, how to measure it, and how to use the results to analyze, evaluate, and ultimately improve processes and products.

Communication is the final aspect. All software engineering activities include written and oral communication. Most produce documentation. A software engineer must have good general technical communication skills, as well as an understanding of forms of documentation appropriate for each activity.

By considering the activity dimension and the aspect dimension as orthogonal, we have a matrix of ideas that might serve as the central thread in a course (Figure A1.1). It is likely that individual cells in the matrix represent too specialized a topic for a full semester course. Therefore, we recommend that courses be designed around part or all of a horizontal or vertical slice through that matrix.

Activities

Development

(requirements analysis, specification, design, implementation, testing, ...)

Control

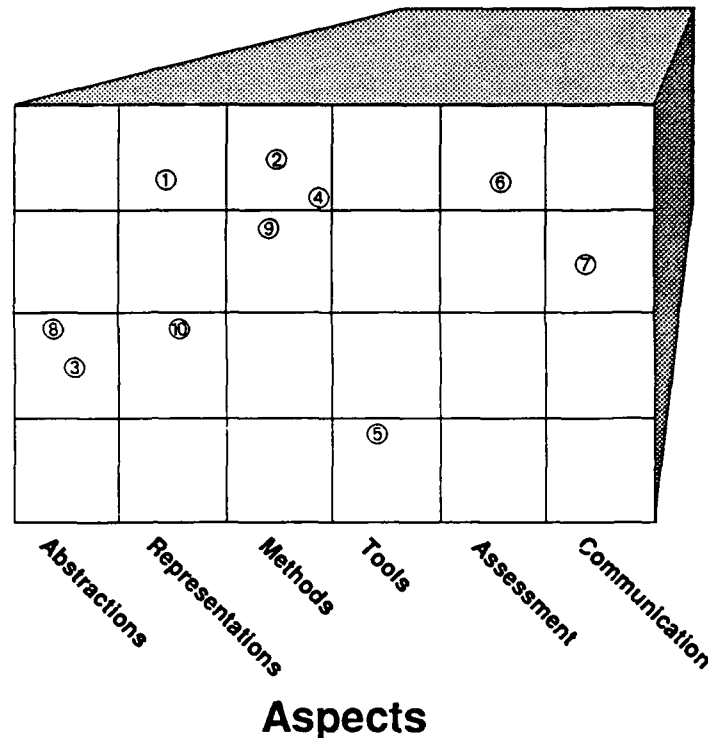
(quality assurance, configuration management, independent V&V, ...)

Management

(project planning, resource allocation, cost estimation, contracting, ...)

Operations

(training, system transition, operation, retirement, ...)



Examples

- | | |
|---------------------------|----------------------------------|
| 1. Ada | 6. Performance Evaluation |
| 2. Object-Oriented Design | 7. Configuration Management Plan |
| 3. COCOMO Model | 8. Waterfall Model |
| 4. Path Coverage Testing | 9. Code Inspection |
| 5. Interactive Video | 10. PERT Chart |

Figure A1.1. The process view: examples of activities and aspects

The Product View

Often it is appropriate to discuss many activities and aspects in the context of a particular kind of software system. For example, concurrent programming has a variety of notations for specification, design, and implementation that are not needed in sequential programming. Instead of inserting one segment or lecture on concurrent programming in each of several courses, it is probably better to gather all the appropriate information on concurrent programming into one course. A similar argument can be made for information related to various system requirements; for example, achieving system robustness involves aspects of requirements definition, specification, design, and testing.

Therefore we have added two additional categories to the curriculum content organizational structure: *software system classes* and *pervasive system requirements*. Although these may

be viewed as being dimensions orthogonal to the activity and aspect dimensions, it is not necessarily the case that every point in the resulting four-dimensional space represents a topic for which there exists a body of knowledge, or for which a course should be taught. Figure A1.2 shows an example of a point for which there is probably a very small but nonempty body of knowledge.

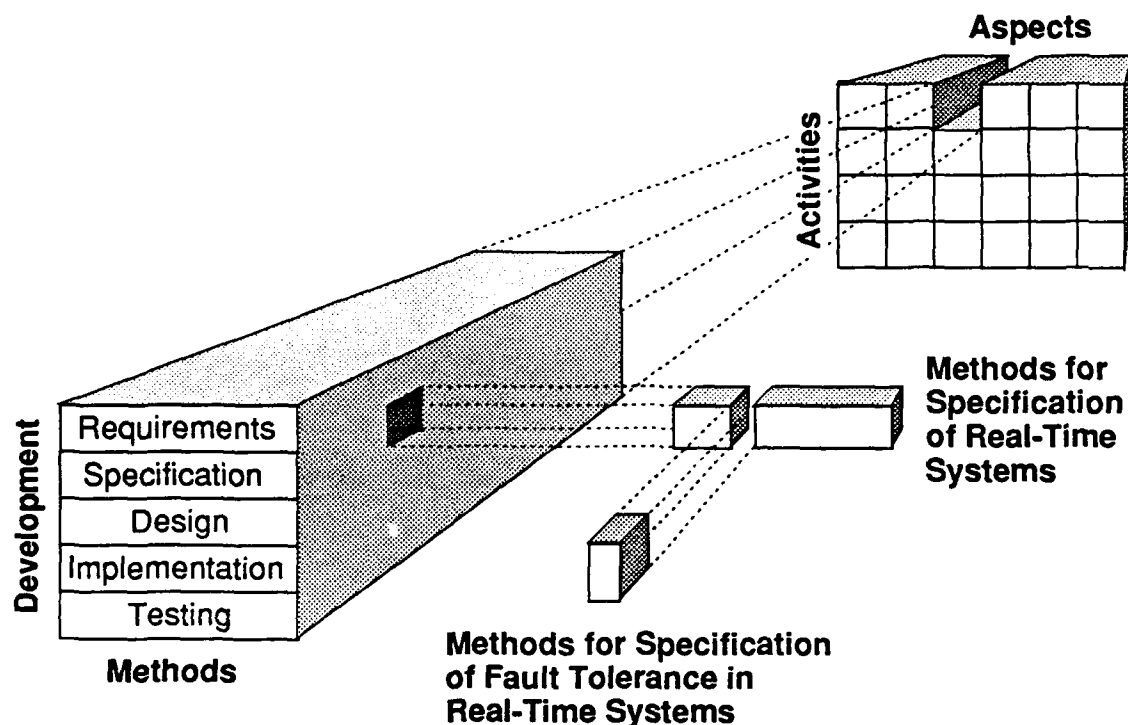


Figure A1.2. Organizational structure for curriculum content

Any of the various system classes or pervasive requirements described below might be the central thread in a course in a software engineering curriculum. We emphasize that the material taught might also be taught in courses whose central thread is one of the activities mentioned earlier. For example, techniques for designing real-time systems could be taught in a design course or in a real-time systems course. Testing methods to achieve system robustness could be taught in a testing course or in a robustness course. The purpose of adding these two new dimensions to the structure is to allow better descriptions of possible courses.

Software System Classes

Several different classes can be considered. One group of classes is defined in terms of a system's relationship to its environment, and has members described by terms such as *batch*,

interactive, reactive, real-time, and embedded. Another group has members described by terms such as *distributed, concurrent, or network.* Another is defined in terms of internal characteristics, such as *table-driven, process-driven, or knowledge-based.* We also include generic or specific applications areas, such as *avionics systems, communications systems, operating systems, or database systems.*

Clearly, these classes are not disjoint. Each class is composed of members that have certain common characteristics, and there is or may be a body of knowledge that directly addresses the development of systems with those characteristics. Thus each class may be the central theme in a software engineering course.

Pervasive System Requirements

Discussions of system requirements generally focus on functional requirements. There are many other categories of requirements that also deserve attention. Identifying and then meeting those requirements is the result of many activities performed throughout the software engineering process. As with system classes, it may be appropriate to choose one of these requirement categories as the central thread for a course, and then to examine those activities and aspects that affect it.

Examples of pervasive system requirements are *accessibility, adaptability, availability, compatibility, correctness, efficiency, fault tolerance, integrity, interoperability, maintainability, performance, portability, protection, reliability, reusability, robustness, safety, security, testability, and usability.* Definitions of these terms may be found in the ANSI/IEEE Glossary of Software Engineering Terminology [IEEE83].

Appendix 2. Bloom's Taxonomy of Educational Objectives

Bloom [Bloom56] has defined a *taxonomy of educational objectives* that describes several levels of knowledge, intellectual abilities, and skills that a student might derive from education (Figure A2.1). This taxonomy can be used to help describe the objectives, and thus the style and depth of presentation, of a software engineering curriculum.

Evaluation: The student is able to make qualitative and quantitative judgments about the value of methods, processes, or artifacts. This includes the ability to evaluate conformance to a standard, and the ability to develop evaluation criteria as well as apply given criteria. The student can also recognize improvements that might be made to a method or process, and to suggest new tools or methods.

Synthesis: The student is able to combine elements or parts in such a way as to produce a pattern or structure that was not clearly there before. This includes the ability to produce a plan to accomplish a task such that the plan satisfies the requirements of the task, as well as the ability to construct an artifact. It also includes the ability to develop a set of abstract relations either to classify or to explain particular phenomena, and to deduce new propositions from a set of basic propositions or symbolic representations.

Analysis: The student can identify the constituent elements of a communication, artifact, or process, and can identify the hierarchies or other relationships among those elements. General organizational structures can be identified. Unstated assumptions can be recognized.

Application: The student is able to apply abstractions in particular and concrete situations. Technical principles, techniques, and methods can be remembered and applied. The mechanics of the use of appropriate tools have been mastered.

Comprehension: This is the lowest level of understanding. The student can make use of material or ideas without necessarily relating them to others or seeing the fullest implications. Comprehension can be demonstrated by rephrasing or translating information from one form of communication to another, by explaining or summarizing information, or by being able to extrapolate beyond the given situation.

Knowledge: The student learns terminology and facts. This can include knowledge of the existence and names of methods, classifications, abstractions, generalizations, and theories, but does not include any deep understanding of them. The student demonstrates this knowledge only by recalling information.

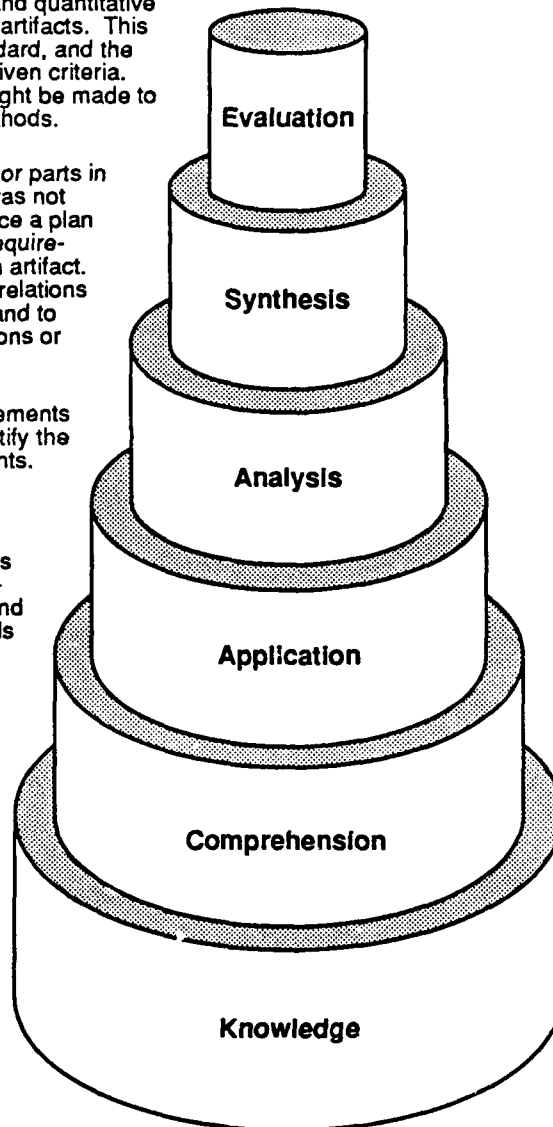


Figure A2.1. Bloom's taxonomy of educational objectives

Appendix 3. SEI Curriculum Modules and Other Publications

The SEI Education Program has produced a variety of educational materials to support software engineering education. The documents listed below (excluding conference proceedings) are available from the SEI; please address *written* requests, accompanied by a mailing label, to the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, Attn.: Donna Mahoney.

Most of the materials described here were written by experienced computer scientists, software engineers, and educators. Because of their expertise, they were invited to spend a period of time at the SEI, where they worked with the Education Program staff to document their special knowledge. The authors of the curriculum modules illustrate the diversity of these contributors over the last four years.

SEI Curriculum Modules and Support Materials

A *curriculum module* documents and explicates a body of knowledge within a relatively small and focused topic area of software engineering. Its major components are a detailed, annotated outline of the topic area, an annotated bibliography of the important literature in the area, and suggestions for teaching the material. It is primarily intended to be used by an instructor in designing and teaching part or all of a course.

A *support materials* package includes a variety of materials helpful in teaching a course, such as examples, exercises, or project ideas. A goal of the SEI Education Program is to provide such a package for each curriculum module. Contributions from software engineering educators are solicited.

The currently available modules and support materials packages are listed below[†]. For each module, a *capsule description*, which is similar to a college catalog description or the abstract of a technical paper, is included.

[†]CM-1 and CM-15 do not appear in this list. CM-1 has been superseded by CM-19, and CM-15 is still under development.

Introduction to Software Design

David Budgen,
University of Stirling

SEI-CM-2-2.1

This curriculum module provides an introduction to the principles and concepts relevant to the design of large programs and systems. It examines the role and context of the design activity as a form of problem-solving process, describes how this is supported by current design methods, and considers the strategies, strengths, limitations, and main domains of application of these methods.

The Software Technical Review Process

James Collofello,
Arizona State University

SEI-CM-3-1.5

This curriculum module consists of a comprehensive examination of the technical review process in the software development and maintenance life cycle. Formal review methodologies are analyzed in detail from the perspective of the review participants, project management and software quality assurance. Sample review agendas are also presented for common types of reviews. The objective of the module is to provide the student with the information necessary to plan and execute highly efficient and cost effective technical reviews.

Support Materials for The Software Technical Review Process

Edited by John Cross,
Indiana University of
Pennsylvania

SEI-SM-3-1.0

This support materials package includes materials helpful in teaching a course on the software technical review process.

Software Configuration Management

James Tomayko,
The Wichita State University

SEI-CM-4-1.3

Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. It emphasizes the importance of configuration control in managing software production.

Support Materials for Software Configuration Management

Edited by James E. Tomayko,
The Wichita State University

SEI-SM-4-1.0

This support materials package includes materials helpful in teaching a course on configuration management.

Information Protection

Fred Cohen,
University of Cincinnati

SEI-CM-5-1.2

This curriculum module is a broad based introduction to information protection techniques. Topics include the history and present state of cryptography, operating system protection, network protection, data base protection, physical security techniques, cost benefit tradeoffs, social issues, and current research trends. The successful student in this course will be prepared for an in-depth course in any of these topics.

Software Safety

Nancy Leveson,
University of California,
Irvine

SEI-CM-6-1.1

Software safety involves ensuring that software will execute within a system context without resulting in unacceptable risk. Building safety-critical software requires special procedures to be used in all phases of the software development process. This module introduces the problems involved in building such software along with the procedures that can be used to enhance the safety of the resulting software product.

Assurance of Software Quality

Brad Brown,
Boeing Military Airplanes

SEI-CM-7-1.1

This module presents the underlying philosophy and associated principles and practices related to the assurance of software quality. It includes a description of the assurance activities associated with the phases of the software development life-cycle (*e.g.*, requirements, design, test, etc.).

Formal Specification of Software

Alfs Berztiss,
University of Pittsburgh

SEI-CM-8-1.0

This module introduces methods for the formal specification of programs and large software systems, and reviews the domains of application of these methods. Its emphasis is on the functional properties of software. It does not deal with the specification of programming languages, the specification of user-computer interfaces, or the verification of programs. Neither does it attempt to cover the specification of distributed systems.

Support Materials for Formal Specification of Software

Edited by Alfs Berztiss,
University of Pittsburgh

SEI-SM-8-1.0

This support materials package includes materials helpful in teaching a course on formal specification of software.

Unit Testing and Analysis

Larry Morell,
College of William and Mary

SEI-CM-9-1.2

This module examines the techniques, assessment, and management of unit testing and analysis. Testing and analysis strategies are categorized according to whether their coverage goal is functional, structural, error-oriented, or a combination of these. Mastery of the material in this module allows the software engineer to define, conduct, and evaluate unit tests and analyses and to assess new techniques proposed in the literature.

Models of Software Evolution: Life Cycle and Process

Walt Scacchi,
University of Southern
California

SEI-CM-10-1.0

This module presents an introduction to models of software system evolution and their role in structuring software development. It includes a review of traditional software life-cycle models as well as software process models that have been recently proposed. It identifies three kinds of alternative models of software evolution that focus attention to either the products, production processes, or production settings as the major source of influence. It examines how different software engineering tools and techniques can support life-cycle or process approaches. It also identifies techniques for evaluating the practical utility of a given model of software evolution for development projects in different kinds of organizational settings.

Software Specification: A Framework

Dieter Rombach,
University of Maryland

SEI-CM-11-1.0

This module provides a framework for specifying software processes and products. The specification of a software product type describes how the corresponding products should look. The specification of a software process type describes how the corresponding processes should be performed.

Software Metrics

Everald Mills,
Seattle University

SEI-CM-12-1.1

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality. This module introduces the most commonly used software metrics and reviews their use in constructing models of the software development process. Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as we gain additional experience with various metrics and models.

Introduction to Software Verification and Validation

James Collofello,
Arizona State University

SEI-CM-13-1.1

Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.

Intellectual Property Protection for Software

Pamela Samuelson and
Kevin Deasy,
University of Pittsburgh
School of Law

SEI-CM-14-2.1

This module provides an overview of the U.S. intellectual property laws that form the framework within which legal rights in software are created, allocated, and enforced. The primary forms of intellectual property protection that are likely to apply to software are copyright, patent, and trade secret laws, which are discussed with particular emphasis on the controversial issues arising in their application to software. A brief introduction is also provided to government software acquisition regulations, trademark, trade dress, and related unfair competition issues that may affect software engineering decisions, and to the Semiconductor Chip Protection Act.

Software Development Using VDM

Jan Storbak Pedersen,
Dansk Datamatik Center

SEI-CM-16-1.0

This module introduces the Vienna Development Method (VDM) approach to software development. The method is oriented toward a formal model view of the software to be developed. The emphasis of the module is on formal specification and systematic development of programs using VDM. A major part of the module deals with the particular specification language (and abstraction mechanisms) used in VDM.

User Interface Development

Gary Perlman,
Ohio State University

SEI-CM-17-1.0

This module covers the issues, information sources, and methods used in the design, implementation, and evaluation of *user interfaces*, the parts of software systems designed to interact with people. User interface *design* draws on the experiences of designers, current trends in input/output technology, cognitive psychology, human factors (ergonomics) research, guidelines and standards, and on the feedback from evaluating working systems. User interface *implementation* applies modern software development techniques to building user interfaces. User interface *evaluation* can be based on empirical evaluation of working systems or on the predictive evaluation of system design specifications.

Support Materials for User Interface Development

Edited by Gary Perlman,
Ohio State University

SEI-SM-17-1.0

This support materials package includes materials helpful in teaching a course on user interface development.

An Overview of Technical Communication for the Software Engineer

Robert Glass,
Computing Trends, Inc.

SEI-CM-18-1.0

This module presents the fundamentals of technical communication that might be most useful to the software engineer. It discusses both written and oral communication.

Software Requirements

John Brackett,
Boston University
SEI-CM-19-1.0

This curriculum module is concerned with the definition of software requirements—the software engineering process of determining what is to be produced—and the products generated in that definition. The process involves: (1) requirements identification, (2) requirements analysis, (3) requirements representation, (4) requirements communication, and (5) development of acceptance criteria and procedures. The outcome of requirements definition is a precursor of software design.

Formal Verification of Programs

Alfs Berztiss,
University of Pittsburgh;
Mark Ardis, SEI
SEI-CM-20-1.0

This module introduces formal verification of programs. It deals primarily with proofs of sequential programs, but also with consistency proofs for data types and deduction of particular behaviors of programs from their specifications. Two approaches are considered: verification after implementation that a program is consistent with its specification, and parallel development of a program and its specification. An assessment of formal verification is provided.

Software Project Management

James E. Tomayko,
The Wichita State University;
Harvey K. Hallman, SEI
SEI-CM-21-1.0

Software project management encompasses the knowledge, techniques, and tools necessary to manage the development of software products. This curriculum module discusses material that managers need to create a plan for software development, using effective estimation of size and effort, and to execute that plan with attention to productivity and quality. Within this context, topics such as risk management, alternative life-cycle models, development team organization, and management of technical people are also discussed.

Selected SEI Educational Support Materials

Teaching a Project-Intensive Introduction to Software Engineering

James E. Tomayko
CMU/SEI-87-TR-20

This report is meant as a guide to the teacher of the introductory course in software engineering. It contains a case study of a course based on a large project. Other models of course organization are also discussed. Appendices A-Z of this report contain materials used in teaching the course and the complete set of student-produced project documentation. These are available for \$55.00 (\$20.00 for the first copy sent to an Academic Affiliate institution).

Software Maintenance Exercises for a Software Engineering Project Course

Charles B. Engle, Jr.,
Gary Ford, Tim Korson
CMU/SEI-89-EM-1

This report provides an operational software system of 10,000 lines of Ada and several exercises based on that system. Concepts such as configuration management, regression testing, code reviews, and stepwise abstraction can be taught with these exercises. Diskettes containing code and documentation may be ordered for \$10.00. (Please request either IBM PC or Macintosh disk format.)

Conference Proceedings

The conference and workshop records below are available directly from Springer-Verlag. Prices are indicated. Please send orders directly to the publisher: Book Order Fulfillment, Springer-Verlag New York, Inc., Service Center Secaucus, 44 Hartz Way, Secaucus, NJ 07094. The numbers shown are ISBNs. Please specify these when ordering.

Software Engineering Education: The Educational Needs of the Software Community

Norman E. Gibbs and
Richard E. Fairley, editors
ISBN 0-387-96469-X

This volume contains the extended proceedings of the 1986 Software Engineering Education Workshop, held at the SEI and sponsored by the SEI and the Wang Institute of Graduate Studies. This workshop of invited software engineering educators focused on master's level education in software engineering, with some discussion of undergraduate and doctoral level issues. Hardback, \$32.00.

Issues in Software Engineering Education: Proceedings of the 1987 SEI Conference

Richard Fairley and
Peter Freeman, editors
ISBN 3-540-96840-7

Proceedings of the 1987 SEI Conference on Software Engineering Education, held in Monroeville, Pa. Hardback, \$45.00.

Software Engineering Education: SEI Conference 1988

Gary Ford, editor
ISBN 3-540-96854-7

Proceedings of the 1988 SEI Conference on Software Engineering Education, held in Fairfax, Va. (Lecture Notes in Computer Science No. 327.) Paperback, \$20.60.

Appendix 4. Cumulative Acknowledgements

The curriculum recommendations in this report have benefitted from the efforts of many people. We had valuable discussions with many members of the SEI technical staff, including Mario Barbacci, Maribeth Carpenter, Clyde Chittister, Lionel Deimel, Larry Druffel, Peter Feiler, Priscilla Fowler, Dick Martin, John Nestor, Joe Newcomer, Mary Shaw, Nelson Weiderman, Chuck Weinstock, and Bill Wood, and with visiting staff members Bob Aiken, Alfs Berztiss, John Brackett, Brad Brown, David Budgen, Fred Cohen, Jim Collofello, Chuck Engle, Bob Glass, Paul Jorgensen, Nancy Leveson, Ev Mills, Larry Morell, Dieter Rombach, Rich Sincovec, Joe Turner, and Peggy Wright.

Earlier versions of the MSE recommendations were written by Jim Collofello and Jim Tomayko, and reviewed by Evans Adams, David Barnard, Dan Burton, Phil D'Angelo, David Gries, Ralph Johnson, David Lamb, Manny Lehman, John Manley, John McAlpin, Richard Nance, Roger Pressman, Dieter Rombach, George Rowland, Viswa Santhanam, Walt Scacchi, Roger Smeaton, Joe Touch, and K. C. Wong.

An early version of the MSE curriculum was the subject of discussion at the Software Engineering Education Workshop, which was held at the SEI in February 1986 [Gibbs87]. In addition to several of the people mentioned above, the following participants at the workshop contributed ideas to the current curriculum recommendations: Bruce Barnes, Victor Basili, Jon Bentley, Gordon Bradley, Fred Brooks, James Comer, Dick Fairley, Peter Freeman, Susan Gerhart, Nico Habermann, Bill McKeeman, Al Pietrasanta, Bill Richardson, Bill Riddle, Walter Seward, Ed Smith, Dick Thayer, David Wortman, and Bill Wulf.

The six MSE core courses were developed by Mark Ardis, Jim Collofello, Lionel Deimel, Dick Fairley, Gary Ford, Norm Gibbs, Bob Glass, Harvey Hallman, Tom Kraly, Jeff Lasky, Larry Morell, Tom Piatkowski, Scott Stevens, and Jim Tomayko.

Bibliography

- Amoroso88 Amoroso, S., Kuntz, R., Wheeler, T., and Graff, B. "Revised Graduate Software Engineering Curriculum at Monmouth College." *Software Engineering Education; SEI Conference 1988*, Gary A. Ford, ed. New York: Springer-Verlag, 1988, 70-80.
- BCS89 *Undergraduate Software Engineering Curricula*. British Computer Society and the Institution of Electrical Engineers, Feb. 1989.
- Bloom56 Bloom, B. *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. New York: David McKay, 1956.
- Brackett88 Brackett, J., Kincaid, T., and Vidale, R. "The Software Engineering Graduate Program at the Boston University College of Engineering." *Software Engineering Education; SEI Conference 1988*, Gary A. Ford, ed. New York: Springer-Verlag, 1988, 56-63.
- Budgen86 Budgen, D., Henderson, P., and Rattray, C. "Academic/Industrial Collaboration in a postgraduate MSc course in Software Engineering." *Software Engineering Education: The Educational Needs of the Software Community*, Norman E. Gibbs and Richard E. Fairley, eds. New York: Springer-Verlag, 1986, 201-211.
- Collofello82 Collofello, J. S. "A Project-Unified Software Engineering Course Sequence." *Proc. Thirteenth SIGCSE Technical Symposium on Computer Science Education*. 1982, 13-19.
- Comer86 Comer, J. R., and Rodjak, D. J. "Adapting to Changing Needs: A New Perspective on Software Engineering Education at Texas Christian University." *Software Engineering Education: The Educational Needs of the Software Community*, Norman E. Gibbs and Richard E. Fairley, eds. New York: Springer-Verlag, 1986, 149-171.
- Engle89 Engle, C. B., Jr., Ford, G., and Korson, T. *Software Maintenance Exercises for a Software Engineering Project Course*. Educational Materials CMU/SEI-89-EM-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Feb. 1989.
- Ford87 Ford, G., Gibbs, N., and Tomayko, J. *Software Engineering Education: An Interim Report from the Software Engineering Institute*. Tech. Rep. CMU/SEI-87-TR-8, DTIC: ADA 182003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., May 1987.
- Gibbs87 *Software Engineering Education: The Educational Needs of the Software Community*. Norman E. Gibbs and Richard E. Fairley, eds. New York: Springer-Verlag, 1987.
- Horning76 Horning, J. J. "The Software Project as a Serious Game." *Software Engineering Education: Needs and Objectives: Proceedings of an Interface Workshop*, Anthony Wasserman and Peter Freeman, eds. New York: Springer-Verlag, 1976, 71-75.
- IEEE83 *IEEE Standard. Glossary of Software Engineering Terminology*. ANSI/IEEE Std 729-1983, IEEE, 1983.

- Lehman86 Lehman, M. M. "The Software Engineering Undergraduate Degree at Imperial College, London." *Software Engineering Education: The Educational Needs of the Software Community*, Norman E. Gibbs and Richard E. Fairley, eds. New York: Springer-Verlag, 1986, 172-181.
- Mills86 Mills, E. "The Master of Software Engineering [MSE] Program At Seattle University After Six Years." *Software Engineering Education: The Educational Needs of the Software Community*, Norman E. Gibbs and Richard E. Fairley, eds. New York: Springer-Verlag, 1986, 182-200.
- Norman88 Norman, D. A. *The Psychology of Everyday Things*. New York: Basic Books, Inc., 1988.
- NRC85 National Research Council, Commission on Engineering and Technical Systems. *Engineering Education and Practice in the United States: Foundations of Our Techno-Economic Future*. Washington, D.C.: National Academy Press, 1985.
- Scacchi86 Scacchi, W. "The Software Engineering Environment for the System Factory Project." *Proc. Nineteenth Hawaii Intl. Conf. Systems Sciences*, 1986, 822-831.
- Wang86 *Bulletin, School of Information Technology 1986-1987*. Wang Institute of Graduate Studies, July 1986.
- Webster83 *Webster's Ninth New Collegiate Dictionary*. Springfield, Mass.: Merriam-Webster Inc., 1983.
- Wortman86 Wortman, D. B. "Software Projects in an Academic Environment." *Software Engineering Education: The Educational Needs of the Software Community*, Norman E. Gibbs and Richard E. Fairley, eds. New York: Springer-Verlag, 1986, 292-305.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS NONE		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-TR-21			5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-89-29		
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.		6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE		
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213			7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANS COM AIR FORCE BASE HANS COM, MA 01731		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE		8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003		
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO. 63752F	PROJECT NO. N/A	TASK NO. N/A
11. TITLE (Include Security Classification) 1989 SEI REPORT ON GRADUATE SOFTWARE ENGINEERING EDUCATION					
12. PERSONAL AUTHOR(S) Mark Ardis and Gary Ford					
13a. TYPE OF REPORT FINAL		13b. TIME COVERED FROM TO		14. DATE OF REPORT (Yr., Mo., Day) June 1989	
15. PAGE COUNT 86					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	education Master of Software Engineering graduate curriculum		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This annual report on graduate software engineering education describes recent SEI educational activities, including the 1988 SEI Curriculum Design Workshop. A model curriculum for a professional Master of Software Engineering degree is presented, including detailed descriptions of six core courses. Fifteen university graduate programs in software engineering are surveyed.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION		
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER			22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630		22c. OFFICE SYMBOL SEI JPO